



# **SWIFTBOT: AN AUTONOMOUS COURIER ROBOT FOR INDOOR DELIVERIES**

**Syed Hamza Azeem**

**Sahran Riaz**

**Sameer Anees Jaliawala**

**Syed Ahsan Ahmed**

**Sarfraz Shahid Hussain**

**HABIB UNIVERSITY**

**KARACHI, PAKISTAN**

**2020**



# **SWIFTBOT: AN AUTONOMOUS COURIER ROBOT FOR INDOOR DELIVERIES**

The Capstone Design Project  
presented to the academic faculty

by

Syed Hamza Azeem

Sahran Riaz

Sameer Anees Jaliawala

Syed Ahsan Ahmed

Sarfraz Shahid Hussain

in partial fulfillment of the requirements for  
BS Electrical Engineering  
in the  
Dhanani School of Science and Engineering

Habib University

June 2020



This work is licensed under a Creative Commons Attribution 3.0 License.

# **SWIFTBOT: AN AUTONOMOUS COURIER ROBOT FOR INDOOR DELIVERIES**

This capstone design project was advised by:

---

Engr. Junaid Ahmed Memon  
Faculty of Electrical Engineering  
*Habib University*

---

Dr. Muhammad Taj Khan  
Faculty of Computer Science  
*Habib University*

Approved by the Faculty of Electrical Engineering on June 20, 2020.

”As I review the events of my past life I realize how subtle are the influences that  
shape our destinies.”

*Nikola Tesla*

For the frontline workers, survivors, and lockdown victims amid COVID-19, this project aims to be a part of the collective effort to minimize human-to-human contact.

It is designed to facilitate the frontline health workers with the safe and hygienic delivery of medical supplies. Moreover, it is also tuned to facilitate those who are quarantined at home, with the delivery of essential items.

We are all in this together, and we shall overcome. The completion of this project would not have been possible without the sheer dedication and commitment shown by the team members. The love and support from our families and supervisors, who have lifted us up during these depressing and tiring times, are the key determinants of our success.

## **ACKNOWLEDGEMENTS**

We want to acknowledge Engr. Junaid Ahmed Memon for his guidance, support, troubleshooting efforts, progress inquiries, and appreciation throughout the course of the project.

We want to acknowledge Dr. Muhammad Taj Khan for his constant encouragement and efforts to deal with various technical and software related issues.

We would like to acknowledge Mr. Ahmed Bilal for his help and support in all capstone related matters.

We would like to acknowledge Mr. Zeeshan Nafees for his guidance and support in various matters.

We would like to acknowledge Mr. Irfan for his help in mechanical work.

We would like to acknowledge Muhammad Saad Saleem for his constant support and help with prototype design choices.

We want to thank Waleed Bin Khalid and his NERC team for providing us with the base of our first prototype.

We want to acknowledge our families for their love, support, motivation, and appreciation throughout these trying times.

## TABLE OF CONTENTS

<b>List of Tables</b> . . . . .	xi
<b>List of Figures</b> . . . . .	xv
<b>Chapter 1: Introduction and Background</b> . . . . .	1
1.1 Introduction . . . . .	1
1.2 Literature Review . . . . .	2
1.3 Problem Statement/Objective . . . . .	4
<b>Chapter 2: Design Process</b> . . . . .	6
2.1 Clients/Stakeholders and their Requirements . . . . .	6
2.2 Design Concepts . . . . .	8
2.3 Society, Economic, and Ethical considerations . . . . .	11
2.4 Environment and Sustainability Considerations . . . . .	11
2.5 Technical Requirements . . . . .	12
2.5.1 EE Aspects . . . . .	13
2.5.2 EE/CS Overlapping Aspects . . . . .	13
2.5.3 CS Aspects . . . . .	14
<b>Chapter 3: Design Details</b> . . . . .	15
3.1 Solution Statement . . . . .	15
3.2 Solution Overview . . . . .	16



3.3	Mechanical Architecture . . . . .	18
3.3.1	Wheeled Mobile Robots . . . . .	18
3.3.2	Drive Mechanisms . . . . .	18
3.3.3	First Prototype . . . . .	19
3.3.4	Final Prototype . . . . .	20
3.3.5	Structural Components . . . . .	24
3.4	Electrical Architecture . . . . .	25
3.4.1	Electrical System Block Diagram . . . . .	25
3.4.2	Final Prototype Wiring . . . . .	26
3.5	Electrical Components (Working, Applications, and Specifications) . . .	27
3.5.1	RP LIDAR A2 . . . . .	27
3.5.2	Pololu 70:1 Gear Ratio Encoder Motors . . . . .	28
3.5.3	VNH5019 Dual Motor Driver . . . . .	30
3.5.4	Solenoid Lock with Relay Module . . . . .	30
3.5.5	Lithium Ion Battery . . . . .	31
3.5.6	Power Bank . . . . .	31
3.5.7	Raspberry Pi 3 Camera Module . . . . .	32
3.5.8	Raspberry Pi 3 . . . . .	32
3.5.9	Arduino Mega 2560 . . . . .	34
3.6	ROS . . . . .	35
3.6.1	Robot Operating System (ROS) . . . . .	35
3.6.2	Why ROS? . . . . .	35
3.6.3	ROS Concepts . . . . .	36
3.6.4	ROS Setup . . . . .	38

3.6.5	ROS Packages Used . . . . .	39
3.7	Facial Recognition . . . . .	45
3.7.1	Purpose and Overview . . . . .	45
3.7.2	Procedure . . . . .	46
3.8	Android Application . . . . .	48
3.8.1	About . . . . .	48
3.8.2	Signup and Login Page . . . . .	48
3.8.3	Facial Registration and Default Location . . . . .	49
3.8.4	User Dashboard and Delivery History . . . . .	50
3.8.5	Booking Details and Pickup Point . . . . .	51
3.8.6	Add Receiver and About Page . . . . .	52
3.9	Web Application . . . . .	53
3.9.1	About . . . . .	53
3.9.2	Login Page . . . . .	53
3.9.3	Dashboard Page . . . . .	54
3.9.4	Users Table . . . . .	54
3.9.5	Floor Map Editor . . . . .	55
3.9.6	Export Data . . . . .	56
3.10	Servers . . . . .	57
3.10.1	Central Server . . . . .	57
3.10.2	Load Balancing Server . . . . .	57
<b>Chapter 4: Prototyping and Testing . . . . .</b>		<b>58</b>
4.1	Motors . . . . .	58
4.2	PID Control . . . . .	63

4.3 Mapping . . . . .	72
4.4 Laser Scanner Simulation . . . . .	77
<b>Chapter 5: Conclusion and Future Work . . . . .</b>	<b>80</b>
5.1 Conclusion . . . . .	80
5.2 Future Work . . . . .	81
<b>Appendix A: Bill of Materials . . . . .</b>	<b>84</b>
<b>Appendix B: ROS (Robot Operating System) Tutorial . . . . .</b>	<b>85</b>
<b>Appendix C: Facial Recognition Tutorial . . . . .</b>	<b>89</b>
<b>Appendix D: Android Application Tutorial . . . . .</b>	<b>91</b>
<b>Appendix E: First Prototype Kinematics Simulations on MATLAB . . . . .</b>	<b>93</b>
<b>Appendix F: Arduino Code . . . . .</b>	<b>99</b>
F.1 Motor RPM Measurement Code . . . . .	99
F.2 Arduino Based Proportional-Integral Controller . . . . .	101
F.3 Arduino Code Interfaced with ROS . . . . .	105
<b>Appendix G: Python Scripts . . . . .</b>	<b>111</b>
G.1 Python Script for Data Formatting . . . . .	111
G.2 Main Loop . . . . .	111
G.3 open_lock.py . . . . .	118
G.4 call_this.py . . . . .	119
<b>Appendix H: ROS Launch and Configuration Files . . . . .</b>	<b>122</b>
H.1 base_local_planner_params.yaml . . . . .	122

H.2	costmap_common_params.yaml . . . . .	122
H.3	global_costmap_params.yaml . . . . .	123
H.4	local_costmap_params.yaml . . . . .	124
H.5	trajectory_planner.yaml . . . . .	124
H.6	my_robot_configuration.launch . . . . .	125
H.7	move_base.launch . . . . .	127
H.8	simple_navigation_goals.cpp . . . . .	128
<b>Appendix I: 2D Laser Scanner Generated Maps . . . . .</b>		<b>131</b>
<b>References . . . . .</b>		<b>137</b>
<b>Vita . . . . .</b>		<b>138</b>

## LIST OF TABLES

1.1	Characteristics of Some Notable Autonomous Delivery Robots . . . . .	4
3.1	RP LIDAR A2 Specifications [18] . . . . .	28
3.2	Pololu 70:1 Gear Ratio Encoder Motor Specifications [19] . . . . .	28
3.3	VNH5019 Dual Motor Driver Specifications [20] . . . . .	30
3.4	Raspberry Pi 3 Camera Module Specifications [21] . . . . .	32
3.5	Raspberry Pi 3 Specifications [22] . . . . .	34
3.6	Arduino Mega 2560 Specifications [23] . . . . .	35
4.1	Average Speed Difference - Motor Driver . . . . .	62
4.2	Average Speed Difference - DC Supply . . . . .	63
4.3	PID Parameter Tuning Values . . . . .	68

## LIST OF FIGURES

2.1	Brainstorming Map of System . . . . .	9
2.2	High Level System Diagram . . . . .	12
3.1	Overall System Block Diagram . . . . .	16
3.2	Program Flow Diagram . . . . .	17
3.3	First Prototype . . . . .	19
3.4	Final Prototype First Variant . . . . .	20
3.5	Bottom Frame . . . . .	21
3.6	Top Frame . . . . .	22
3.7	Internal View . . . . .	23
3.8	Side View . . . . .	23
3.9	Elevated View . . . . .	24
3.10	Electrical System Block Diagram . . . . .	25
3.11	Overall Wiring . . . . .	26
3.12	Wiring Associated with Veroboard 1 and Veroboard 2 . . . . .	26
3.13	Working of a Quadrature Encoder . . . . .	29
3.14	The tf Transformed Coordinate Frame for a Differential Drive Robot . .	41
3.15	Visual Representation of System Transform Tree . . . . .	42
3.16	An Overview of Gmapping, Hector SLAM, and Google Cartographer .	44
3.17	Mapping Results of Gmapping, Hector SLAM, and Google Cartographer	44

3.18 High Level Overview of Google Cartographer Package . . . . .	45
3.19 Block Diagram of Facial Recognition Flow . . . . .	47
3.20 Block Diagram of Facial Recognition Module . . . . .	48
3.21 Signup Page . . . . .	49
3.22 Login Page . . . . .	49
3.23 Register Face . . . . .	50
3.24 Add Default Location . . . . .	50
3.25 Dashboard . . . . .	51
3.26 History . . . . .	51
3.27 Booking Details . . . . .	52
3.28 Pickup Point . . . . .	52
3.29 Add Receiver Page . . . . .	53
3.30 Admin Login . . . . .	54
3.31 Admin Dashboard . . . . .	54
3.32 Users Table . . . . .	55
3.33 Floor Map Editor . . . . .	56
3.34 Export Data . . . . .	56
3.35 Export to Excel . . . . .	57
4.1 Encoder Pulse Output . . . . .	59
4.2 Motor RPMs Over Time . . . . .	60
4.3 puTTY Configuration for Serial Data Collection . . . . .	61
4.4 Motor Speed Comparison - Motor Driver . . . . .	62
4.5 Motor Speed Comparison - DC Supply . . . . .	62
4.6 PID Parameters . . . . .	65

4.7	Map of Test Area for Prototype 1 . . . . .	66
4.8	Right Wheel Target and Actual Speeds . . . . .	67
4.9	Left Wheel Target and Actual Speeds . . . . .	67
4.10	PID Tuning Result 1 . . . . .	69
4.11	PID Tuning Result 2 . . . . .	69
4.12	PID Tuning Result 3 . . . . .	70
4.13	PID Tuning Result 4 . . . . .	70
4.14	PID Tuning Result 5 . . . . .	71
4.15	Final PID Tuning Result . . . . .	72
4.16	Map of Projects Lab . . . . .	74
4.17	Ineffective Map . . . . .	75
4.18	Final Map . . . . .	76
4.19	Final Demonstration Start and End Points . . . . .	77
4.20	Robot Model in Simulator . . . . .	78
4.21	180° Scan with Obstacles - Gazebo and Rviz Visualizations . . . . .	78
4.22	360° Scan with Obstacles - Gazebo and Rviz Visualizations . . . . .	79
A.1	SWIFTBOT - Bill of Materials . . . . .	84
E.1	Linear Trajectory . . . . .	93
E.2	Elliptical Trajectory . . . . .	94
E.3	Pose for Inverse Kinematics . . . . .	95
E.4	Kinematic Control Inputs Given Pose . . . . .	96
E.5	Obstacle Avoidance and Go-to-Goal Behavior . . . . .	97
E.6	Control Inputs Using Inverse Kinematics . . . . .	98



I.1	Projects Lab Map 1 . . . . .	131
I.2	Projects Lab Map 2 . . . . .	132
I.3	House Map . . . . .	133

## EXECUTIVE SUMMARY

This document is produced for the purpose of highlighting the details of the capstone project that team SwiftBot brings to fruition – the design and implementation of an autonomous delivery robot for indoor deliveries, and payload transportation. The main problem that this project seeks to address is the safe, secure, and timely delivery of small payloads, such as, food items, logistical supplies and equipment, and documents in indoor and closed campus spaces, such as, offices, university campuses, and hospital environments. The final prototype is demonstrable for a single floor plan, such as a large indoor space, and it is integrated with an app to coordinate and manage deliveries linked to the robot, via a central server. Some of the main functionalities which can be found in the final prototype include 2D environment mapping, obstacle detection and avoidance, and autonomous navigation and path planning behavior. The major challenges involved in implementing a functional project of this caliber ranges from effective hardware implementation – making use of limited resources to ensure a smoothly functioning final product – to software and hardware integration in an efficient and time considerate manner, not to mention the learning curve associated with some of the project subsystems. Ensuring smooth movement of the robot is also an area that can be worked towards. However, with dedication and commitment, the team delivers a project that contributes towards the application domain of service robotics. The way forward with this project would be to incorporate some form of sensor fusion to better perceive the environment.

**KEYWORDS:** Autonomous Robot, Service Robotics, Indoor Deliveries

# **CHAPTER 1**

## **INTRODUCTION AND BACKGROUND**

### **1.1 Introduction**

The need for secure and 'swift' delivery has risen to the forefront in today's increasingly connected society, a society that relies heavily on delivery services. From local food deliveries to worldwide shipping, the reliability and safety of delivery services can significantly impact the efficacy of operations involving the movement of goods. Another revolution being experienced is in the increasing degree of automation the world is undergoing which has altered the landscape of many traditional industries, including that of delivery logistics. Be it in factories, offices, or hospitals, issues related to the transportation of items within organizations exist that can hinder or delay tasks which are to be completed within fixed time frames and hence, there exists a requirement for tackling the issue of deliveries within closed spaces in a safe and timely manner.

This project puts forth an autonomous unmanned ground vehicle that utilizes localization and mapping capabilities to plan a path to a destination intended for the delivery of a payload. A mobile application integrated with a mapping service is utilized to mark a destination area or coordinate. Moreover, via the data harvested from a laser range sensor responsible for object detection, the mapping and path planning process is used to automate the delivery process between two points. Security features such as facial recognition are possible means to secure the payload within the robot.

Part of the motivation of the project stems from observing the transportation needs on Habib University's campus, from transporting food items to and from the cafeteria, to moving exam papers and lab manuals between different offices and departments when dealing with bulk loads and downed printing services. With the team's interest in robotics and the desire to make the campus life easier for its denizens, potential use cases were brainstormed that focused on providing a solution to the aforementioned

problems. All these factors led to the team going forward with the project.

## **1.2 Literature Review**

The motivation of the project initially stemmed from observing efficiency and convenience factors of transportation on Habib University's campus, such as, for food deliveries from the cafeteria and moving equipment between labs in a safe and rapid manner. After literature surveys, the transportation problem was also identified in other scenarios, such as, in hospital settings where time-critical deliveries, such as, lab sample delivery, medical instrument delivery etc. are required [1] [2]. Existing solutions that focus on automating hospital logistics via robots include the HelpMate robot [3] which was commissioned in the late 90s. Besides possessing a dated design, the cost of purchasing this robot lay around USD 110,000 with weekly rentals costing USD 25,000 making this solution accessible to a select few large organizations.

Surveys also led to the discovery of other attempts at tackling similar delivery areas as those highlighted above. E-commerce sales are steadily increasing throughout the world. In the United States, as indicated by the United States Census Bureau's Quarterly E-Commerce Report [4], e-commerce sales increased by an average annual rate of 16% in the last twenty years. Concurrently, the acceptable amount of time that people deem for deliveries is also decreasing [5] and these two cases have presented a requirement for delivery organizations to invest in technologies to improve the efficiency, productivity, and the time factor associated with deliveries. Sidewalk Autonomous Delivery Robots (or SADR as the literature calls them) are potential technologies aimed at tackling the aforementioned issues. SADR are classified as near pedestrian sized robots which deliver items to customers without any involvement of a delivery person. Examples of notable implementations of SADR include Amazon's Scout delivery robot [6] and Starship Technologies' robot delivery service working in the area of short range deliveries in particular settings like university campuses [7], neighborhoods, and office spaces whose occupants serve to constitute some of the relevant stakeholders associated with the problem. However, these solutions are contextualized and optimized for the

neighborhoods and spaces they are operating within. Since, a lot of these robots operate on sidewalks, they are also subject to increasing regulation by local authorities, such as, municipalities, police, safety departments, and transportation authorities. Particular concerns regarding the operation of such robots include being a threat to children, senior citizens, and disabled people as they can't maneuver quickly enough to avoid a moving robot [8]. San Francisco's police commander, Robert O'Sullivan raised a concern regarding the safety of autonomous robots operating on streets and sidewalks due to their potential of becoming dangerous projectiles upon being struck by vehicles [9]. Community groups have also criticized the operating area of SADR as being impediments and nuisances to people walking on sidewalks [10].

All these complaints need to be addressed in the design of a solution if efficiency is desired for delivery technologies. However, according to Starship Technologies, most of the pedestrians do not pay attention to its robots and many react positively to their presence. Additionally, the company has not encountered any accidents since deploying its robots [11]. Current implementations of some delivery robot technologies have some conflicting views regarding their deployments and usage, which brings into play an additional inspection related to regulatory environments that goes beyond the aims and scope of this project. However, keeping some of these criteria in mind, restrictions, such as size, weight, and speed limits are imposed on most delivery robots. The characteristics of some of the notable delivery robots are summarized in Table 1.1.

Table 1.1: Characteristics of Some Notable Autonomous Delivery Robots

<b>Company</b>	<b>Weight (lb)</b>	<b>Speed (mph)</b>	<b>Capacity (lb)</b>	<b>Capacity (chambers)</b>	<b>Range (mi)</b>
Starship Technologies	40	4	40	1	4
Domino's DRU	Unknown	12	21	4	12
Dispatch's Carry	Unknown but requires 2 people to lift device	4	100	4	up to 48 mi with 12 hr battery
Thyssenkrupp's TeleRetail	60	35	77	1	10
Marble	80	4	Unknown	1	Unknown

The market for service robotics is also a booming one. A robotics and AI research partnership between Sony and CMU (Carnegie Mellon University) aimed at optimizing the food preparation and delivery process in confined environments [12] is one among a number of examples which highlight the growing investment in service robotics. Another use case highlighting the efficiency of robots for deliveries in closed spaces is exhibited by the use of wheeled mobile robots in a Las Vegas hotel (The Renaissance) [13] which delivers toiletries and sundry items to hotel guests, essentially serving as waiting staff. A lot of the solutions highlighted are also in the testing phase so accurate data regarding the receptiveness of the solutions is not entirely available. All in all, the market for service robotics and the multitude of use cases presents a viable opportunity for working in the space of delivery robots for indoor spaces.

### 1.3 Problem Statement/Objective

In summary, the broader aim of this project is to tackle the problem of untimely and unsecured deliveries within indoor environments through an indigenously developed prototype delivery robot service which would handle small payload deliveries, such as food parcels, small lab materials, and document stacks in confined environments, such as office floors, and campus spaces, in a safe and secure manner. The further vision

with this project is scalability to handle deliveries for larger complexes and on multiple floors but for the purposes of demonstrability of the concept, the robot's operation is being limited to single floors and indoor spaces. The user can primarily interact with the service through a mobile application to manage what item to order, track the progress of a delivery, and monitor other associated delivery stats. Based on the setting and primary users the relevant decorum and codes would need to be followed particularly in the case of more controlled environments like hospitals and industrial offices. But for more informal operational settings such as university campuses, the regulatory requirements are predicted to be more relaxed, if any. It is desirable for the solution to not be too bulky and noisy so as not to disturb the peace and atmosphere of the campus space it is operating in and so that the robot is more easily integrated in a campus dominated by a sizeable human population.

## **CHAPTER 2**

### **DESIGN PROCESS**

#### **2.1 Clients/Stakeholders and their Requirements**

A number of stakeholders including hospitals, office complexes, university campuses, and hotels are identified 1.2. Based on the goal of implementing an autonomous delivery service to make indoor material transportation more efficient, a number of common features were identified which are enumerated as follows:

1. An autonomous system - a robot - which can independently navigate in the premises of operation with a payload.
2. Path planning behavior towards a goal intended for pickup or delivery.
3. Obstacle detection and avoidance so as to avoid hindering users in the environment of operation and ensure unobstructed movement towards a destination.
4. Mobile application based service for coordinating, tracking and managing deliveries.
5. An admin panel for monitoring the delivery agent and item delivery status and an ability to remotely operate the agent.
6. A locking mechanism for physically securing the payload within the delivery agent.
7. Software based security measure for user verification and access to app and payload – e.g. facial recognition feature.

Additionally, some constraints are considered, based on the implementation of an autonomous system in closed spaces with specific payloads in mind.



1. Mechanical design - The mechanical design of the robot body has to be robust enough to support the payload weight, actuators, and connected components so as to last for a large multitude of operational cycles. These cycles would include those involved in the testing phase of the various subsystems of the robot, as well as, trial runs for the finished product. The design should also possess a high degree of modularity so that it is easier to swap out and change components for maintenance purposes, as opposed to remaking and reattaching parts from scratch.
2. Autonomous functionality and software considerations - Given the scope of the project and keeping in mind the cost of the system to the end user, the decision of using paid or open source development software also had to be considered. Due to the popularity of the open source meta-operating system, ROS, as well as, the fact that a lot of support is available for implementing various modules in ROS, it appeared as a desirable choice. However, this support is often available for particular contexts and hardware requirements, and these contexts or hardware elements for the subsystems may not be used in the project. In such scenarios, it may be required to look for alternate programs and dependencies and interface them with the existing implementation. While the necessary software packages and interfacing requirements have been identified, the more complex aspects of the project, such as, autonomous navigation to a destination based on a user's input on a mobile app, is where novel contributions would shine. For certain behavioral flows and decision making aspects of the robot, it becomes necessary to come up with a novel approach to implement the functionality that is desired.
3. Mobile application - Some non-functional requirements which double as constraints for the mobile application were decided upon. The mobile application should be capable enough to handle the load of many users simultaneously without noticeable effects on its performance, navigation, or glitches at a single time. For instance, it should have a way of handling user queues if the robot is booked, and processing for another user. Common User Interface (UI) design methodolo-

gies should be taken into account in order to make the user interface as appealing as possible. An extension of experience design is the responsiveness of the app, which should ideally not take more than 2-5 seconds to load. Most importantly, the app's codebase needs to be scalable so that more features can be added in the app, as required.

4. Cost - The overarching idea is for the delivery system to be applicable in environments like university campuses and office complexes. With this goal in mind, keeping costs to a minimum while ensuring a good quality product is a strong requirement. In a lot of the literature surveys, it was found that similar solutions are often heavy on the budgets of the organizations that adopt them thus limiting these systems to deployment in only a few locations. Therefore, a challenge in minimizing costs is the selection of hardware, including sensors, embedded controllers, actuators, and connecting components, so that the project does not compromise on the accuracy, robustness, and inherent quality of the delivery system while being able to produce an affordable solution for smaller organizations.

## **2.2 Design Concepts**

One of the aims of the project is to work towards introducing an idea that is not usually thought about in the context of Pakistan's delivery industry, and in service robotics. It is a requirement to integrate existing systems and applications of robotics and automation for a use case that is not often explored in literature, but towards which a lot of industries and research labs are gradually working. Another idea behind this project is to implement the desired system in as little cost as possible so that the prototype may be potentially taken to test phases. Similar solutions often prove to be costly endeavors and are built to be highly contextualized within the environment of the adopter. The focus is on making a product that could potentially prove to be a ubiquitous application for closed spaces and multi-building complexes. The primary point to note is that such robots require a large amount of time, as well as, money to be implemented completely. It is also an aim to make the system highly accurate in terms of navigating, detecting

obstacles, and planning the most efficient paths while considering the resources and constraints at its disposal. The ideation phase of the project presented a number of alternative design choices that had to be weighed against each other while keeping in mind the intended functionality, scope, and user requirements of the project.

Figure 2.1 shows an abstract brainstorming map that provides some design concepts and alternatives that were considered for the system.

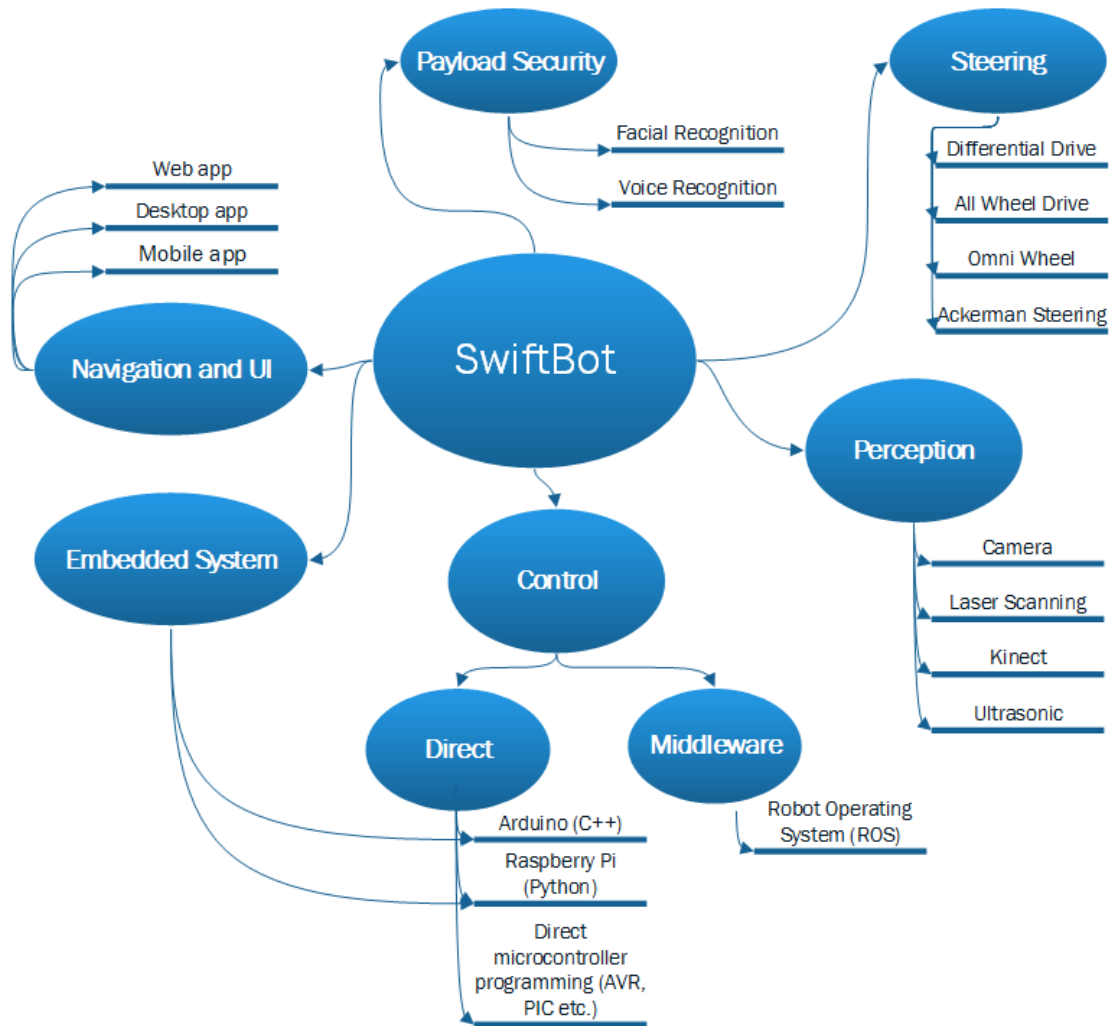


Figure 2.1: Brainstorming Map of System

Points about the design alternatives and considerations are presented as follows:

1. Perception – A number of options exist for sensors to use for localization and mapping. It was thought to go for a 2D perception scheme that is accurate, has readily available software support including systems for error correction, and a

decent range without a sizeable error factor. All these considerations leaned toward using a LIDAR (Light Detection and Ranging) as the mapping tool.

2. Control - This aspect could either be through implementation of behaviors and algorithms directly on microcontroller platforms like the Arduino and Raspberry Pi, or through an intermediary software platform like the Robot Operating System (ROS) which would handle input/output and communication in a more streamlined manner. This 'middleware' has the advantage of making program flow and code management much simpler which is helpful in integration. Since, ROS is open source and possesses an active community support system, it proved to be a natural choice.
3. Steering – A lot of different steering mechanisms were weighed against each other by considering the mechanical requirements and controllability of the robot, and thus the differential drive system was found to be the most appropriate in light of project requirements.
4. Payload security – A number of options for security features exist, for instance, facial recognition, voice recognition, pin, and password encryption in order to make the system impervious to theft and fraudulent attempts. Due to the availability of resources related to computer vision tools, and an observation of increased adoption of camera based biometric security systems, it was decided to apply facial recognition as a viable security measure to secure the payload within the robot during its journey. While facial recognition is not a fool-proof safety mechanism as it can be potentially countered, another reason for selecting this method was to test out a future avenue of using cameras for obstacle detection purposes. Facial recognition is just one aspect of using cameras. If the example of self driving vehicles is taken into consideration, they use multiple sensors for perception so as not to rely on one system; something which increases the chances for failure. Facial recognition, while serving as a security measure in the system, is also an avenue for future exploration of perception using cameras.

5. Embedded system - The choice of the brains of the robot is integral in determining how well the robot can handle the computations it is being subjected to. The Raspberry Pi is an appealing choice due to its comparatively higher computation power, inbuilt communication modules like Bluetooth and WiFi, and its higher memory capabilities as compared to other similarly priced and targeted products in the market.

### **2.3 Society, Economic, and Ethical considerations**

The primary design and system specifications are set according to the target audience that the team is trying to cater to which primarily includes people in a campus space. The prototype of the robot is designed to be as compact as possible and not too wide or tall so as to serve as an impediment and nuisance to pedestrians, and so that it is easier for the system to move around in busy routes. In many newly planned smart cities, there are planned paths for such robots.

The robot has security features, such as, real time tracking and facial recognition, so that the robot can not be stolen, and the contents are confidential to only the intended person. It is a fact that facial recognition poses some potential privacy concerns considering that a database of facial features for users is made for verification purposes. This can be a deterrent for some users who are unsure about their privacy being breached. Therefore, alternative security measures also need to be considered, such as, a pin-code based safety lock to secure the payload, and to identify the user.

### **2.4 Environment and Sustainability Considerations**

The lightweight design of the robot, as well as, its compactness and modularity are selected keeping in mind the power requirements and the need to move efficiently in order to avoid imposing any unnecessary use of power. While there is no metric to assess the carbon footprint incurred by the system in specific terms, the team is using comparisons with other robots of similar size and scope in terms of materials and battery systems used. As an example, the use of multiple layers of reinforced plastic sheets

for the robot's body was considered in order to reduce the weight and to ensure smooth movement without burdening the system's actuators and drawing more power than what is necessary for the robot to move. The materials used are also cheap and commonly available, not requiring any difficult to obtain parts in order to promote reproducibility and scalability. The major chunk of the project cost is borne by the LIDAR sensor of which there are also cheaper varieties available in the market. The model that is used in the project is due to ready availability of the item.

## 2.5 Technical Requirements

A high level system diagram of the system is shown in Figure 2.2. The major components comprising the complete system include the delivery hardware, the perception module, the android application, and the facial recognition server.

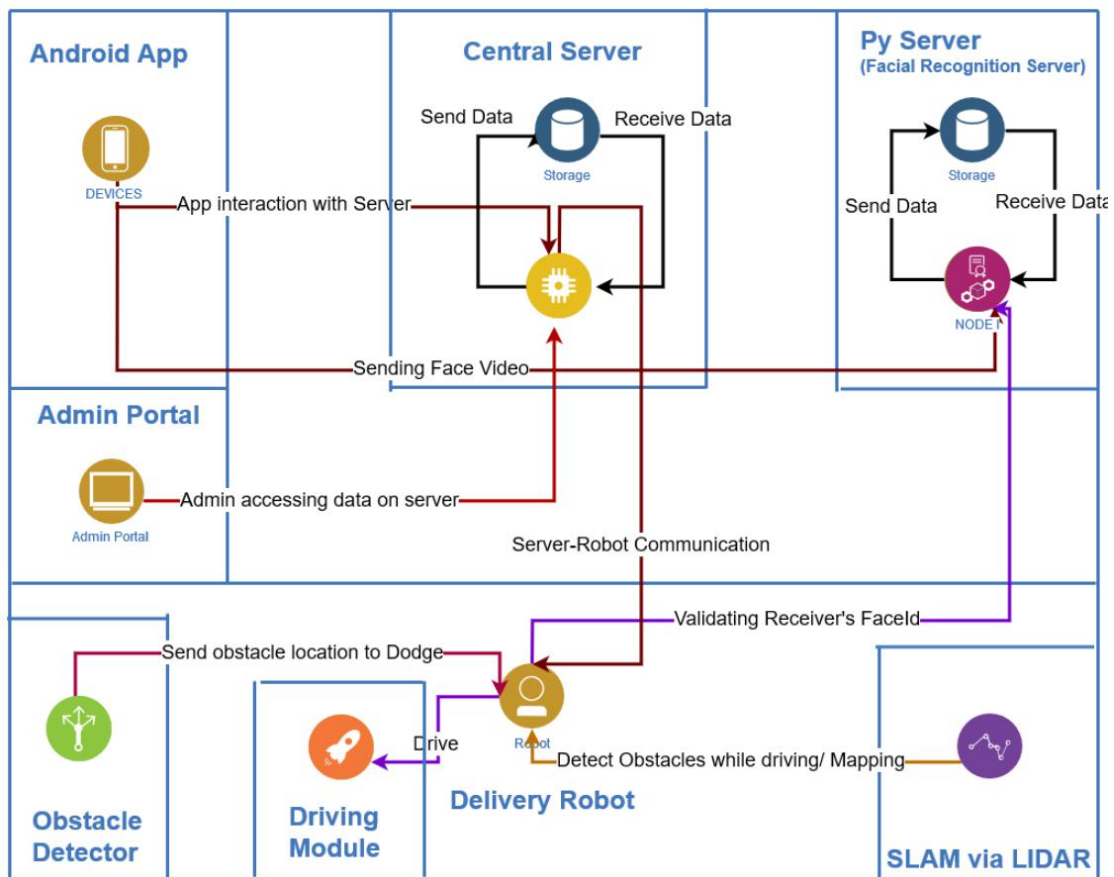


Figure 2.2: High Level System Diagram

The technical requirements of the project are listed as follows:

### 2.5.1 EE Aspects

1. Design of an optimal robot perception system for the task in hand- A LIDAR is the principal component that is being utilized due to its accuracy and reliability.
2. Design and implementation of control system for locomotion – Odometry for speed tracking and PID control for smooth movement.
3. Communication considerations between embedded systems and server – This requires consideration of the specific protocols and wireless systems for communication (e.g. WiFi).
4. Hardware construction: robot model designing and implementation – This involves CAD models, base design, sensor attachments, payload securing attachment, and other hardware attachment design and analysis.

### 2.5.2 EE/CS Overlapping Aspects

1. Space mapping in 2D – The team considered using a laser scanner (LIDAR) for detection of the surrounding as the modules commonly available have built-in actuation mechanisms to rotate the sensor to scan the surrounding in a 2D plane in a power efficient manner with low sound disturbance. This is opposed to cameras or a Kinect sensor for which it would be required to design the rotation mechanism and manipulate the input data to reflect mapping in a 2D plane. The sensor used has a relatively high degree of accuracy, up to a 6 m range.
2. Localization, mapping, and path planning – This includes using the appropriate methods to create a map of the surroundings while simultaneously localizing the robot within the environment being mapped. This would form the basis for the path planning algorithm for delivery to the destination. The ROS operating system has some built in support for SLAM, such as, Hector SLAM and Gmapping methods that also include filters for cleaning the input data that can be configured and optimized to meet the requirements.

3. Obstacle detection and avoidance – This is required to navigate the environment without bumping into obstacles, both static ones, like furniture and walls, as well as, dynamic ones, such as, humans.
4. Remote operation functionality – This would be intended for monitoring purposes (robot location, delivery status etc.) and initially mapping an area manually as required.

### 2.5.3 CS Aspects

1. Client – Server communication implementation for the robot and mobile application to send and receive data and computations.
2. App development for the delivery system and web based admin panel for overarching monitoring and control of operations.
3. Database implementation for holding records and managing data for deliveries and system users.
4. Security system implementation involves using a facial recognition system for security purposes. This includes using computer vision and software frameworks, such as, OpenCV and Python Flask to set up facial recognition features, via video within the mobile app, as well as, on the robot camera, and communicate that information to a central server. Alternatives, such as, password based security, voice recognition, or a combination of these are also considered.



## **CHAPTER 3**

### **DESIGN DETAILS**

#### **3.1 Solution Statement**

The larger goal with this project was to work on a solution for efficiency; in movement, delivery, logistics, and more specifically, in easing the transportation of small goods within closed locations, in a bid to make people's lives easier. Keeping this vision in mind, the design of the solution is undertaken with certain factors in mind to make the proposed product achieve its purpose in a satisfactory manner, while imposing minimal cost to a customer.

The solution involves a low cost autonomous vehicle, paired with a mobile and web app based delivery system, which is optimized for indoor spaces. Most of the hardware and structural materials are locally sourced and selected to minimize the weight and maximize the traction. The use of locally sourced items also helps to reduce the carbon footprint associated with importing and assembling items from different corners of the world. The electronic components and microcontroller systems are selected and utilized keeping in mind the availability, and the cost margin. An open source operating system i.e. ROS is used keeping in mind the potential future development work with a greater technical resource pool available, and readily available community support. An open source system also diminishes the cost involved in building, tailoring, and testing an OS and software suite from scratch. ROS allows a community of developers to work on any potential solutions that may arise, and makes it easier to define a ubiquitous standard for any future development work. The size, weight, and visual design of the robot are taken into consideration so that they provide minimal impedance to the movement of people, while still maintaining a presence as a part of the working environment. The power requirements of the robot are also minimal, primarily in order to minimize the weight associated with high rated batteries. A longer running time is also accounted for as

opposed to many autonomous systems that output powerful capabilities, but ultimately yield to very short operational times. With all these features, the robot is designed to be a sustainable product that can blend easily with the operational environment, and also minimize its cost to the environment with respect to the materials used in its manufacture, as well as, with respect to the impact of its operational activities.

### 3.2 Solution Overview

The overall system and process block diagram of the project is shown in Figure 3.1.

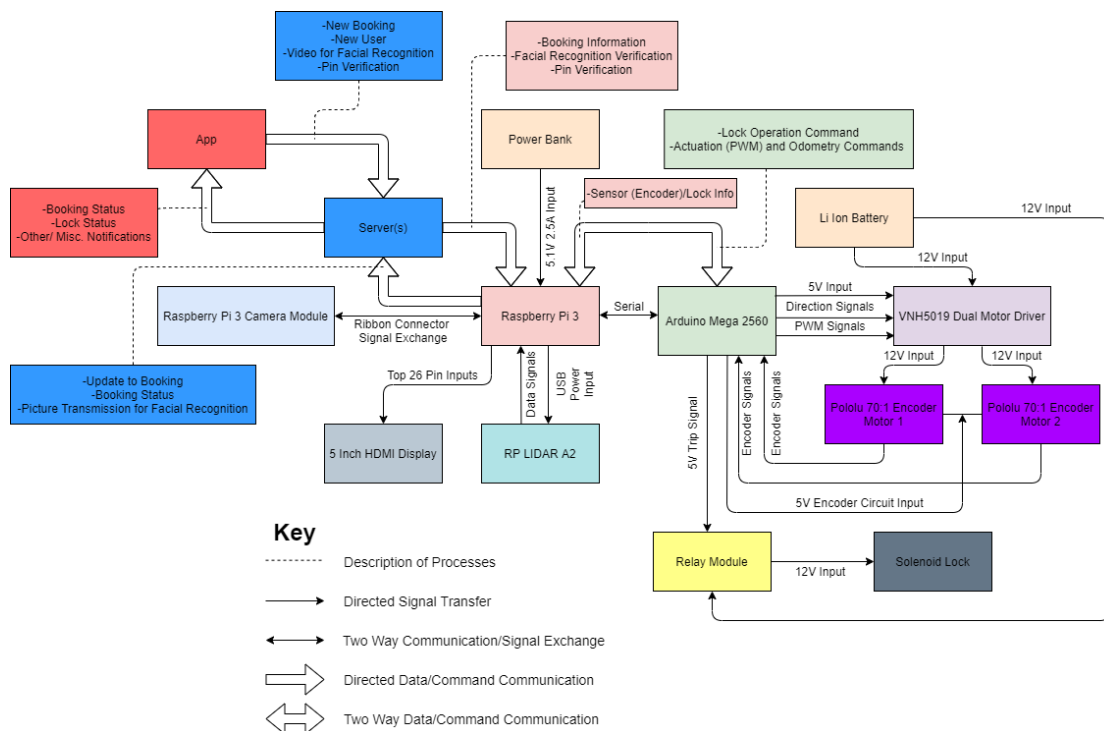


Figure 3.1: Overall System Block Diagram

The program flow diagram is shown in Figure 3.2.

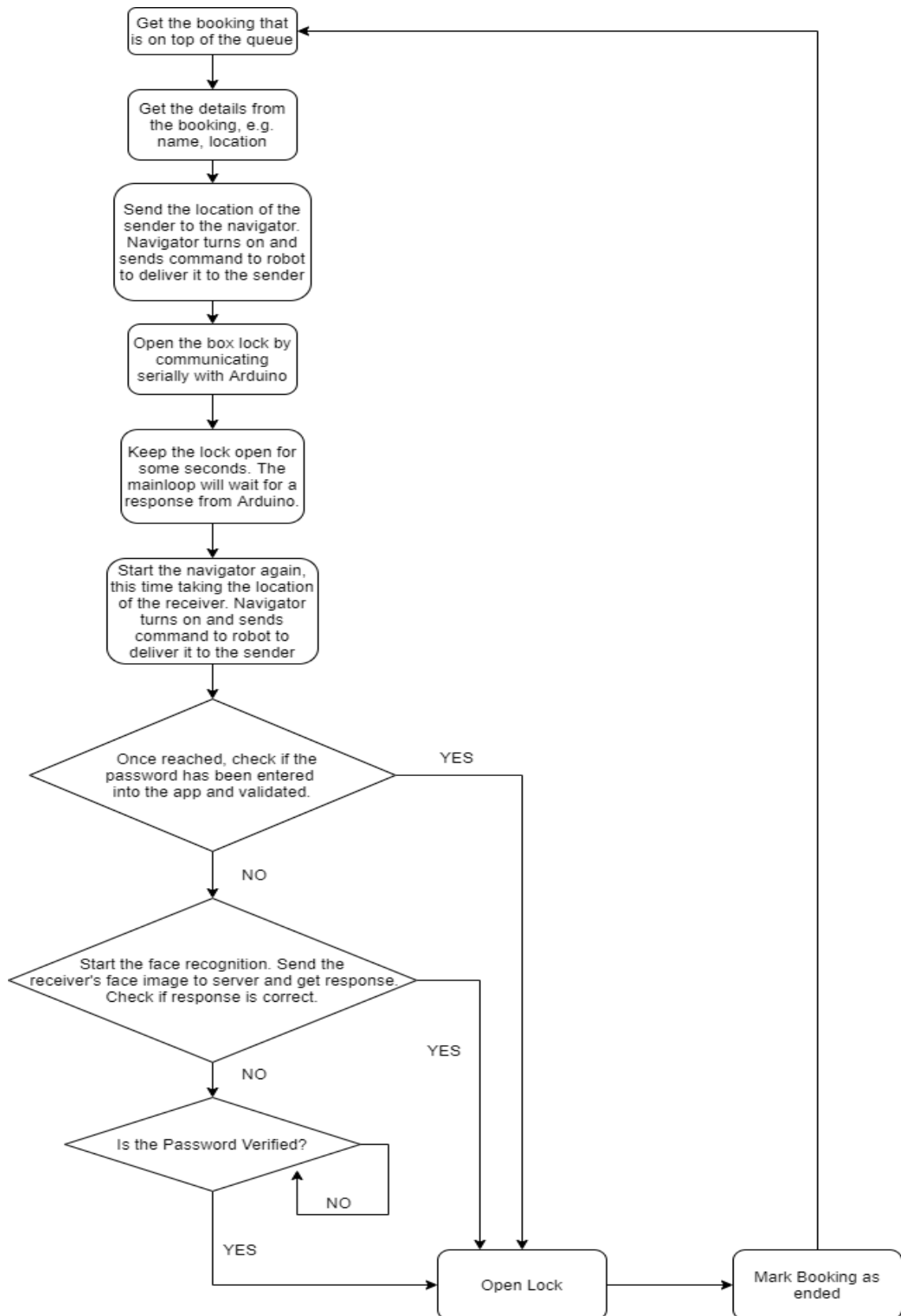


Figure 3.2: Program Flow Diagram

### **3.3 Mechanical Architecture**

#### **3.3.1 Wheeled Mobile Robots**

Robots that navigate around the ground, and use motorized wheels to propel themselves are regarded as wheeled mobile robots [14]. Wheeled mobile robots come with a variety of drive mechanisms [15]. Wheels may be free spinning or drive wheels [15]. They may have a steered axle, a free axle, or a fixed axle [15]. A free axle, free spinning wheel is called a caster [15]. As some drive mechanisms require a pair of wheels, a caster is incorporated to maintain stability on a flat terrain [15]. A disadvantage of wheeled mobile robots is that they cannot navigate well over obstacles, such as, a rocky terrain, sharp declines, or areas with low friction [14]. On the other hand, an advantage is that wheeled mobile robots can accurately observe their environment in real time, as they are equipped with a complex array of sensors to detect surroundings.

#### **3.3.2 Drive Mechanisms**

Differential drive, and Ackermann steering are the major drive mechanisms used in wheeled mobile robots [15]. Differential drive is low cost and simple however, the controls are less precise, designs are generally for lower weight, and it is bad for bumps/obstacles [14] [16]. Differential drive is composed of two wheels that share a fixed axis, and can be driven at different rates to achieve turning [15]. Depending on the speed of rotation, and its direction, the center of rotation may fall anywhere in the line joining the two wheels [14]. Ackermann steering is used by most cars, it is similar to a pair of bicycles – two steered wheels in the front, and two parallel fixed wheels in back [15]. With Ackermann steering, wheels do not need to slip to turn. Moreover, control geometry is made easier via fixed rear wheels [16]. As a drawback, the motor count is increased [16].

The other drive mechanisms studied as a part of the literature review include omni, mecanum, and 2-by-2 powered wheels. These drive mechanisms were disregarded on the basis of their modeling and control complexities, sensitivity to non-smooth terrain,

and power requirements [16]. Taking into consideration the lack of resources for help related to the aforementioned mechanisms, the differential drive mechanism was chosen for this project.

### 3.3.3 First Prototype

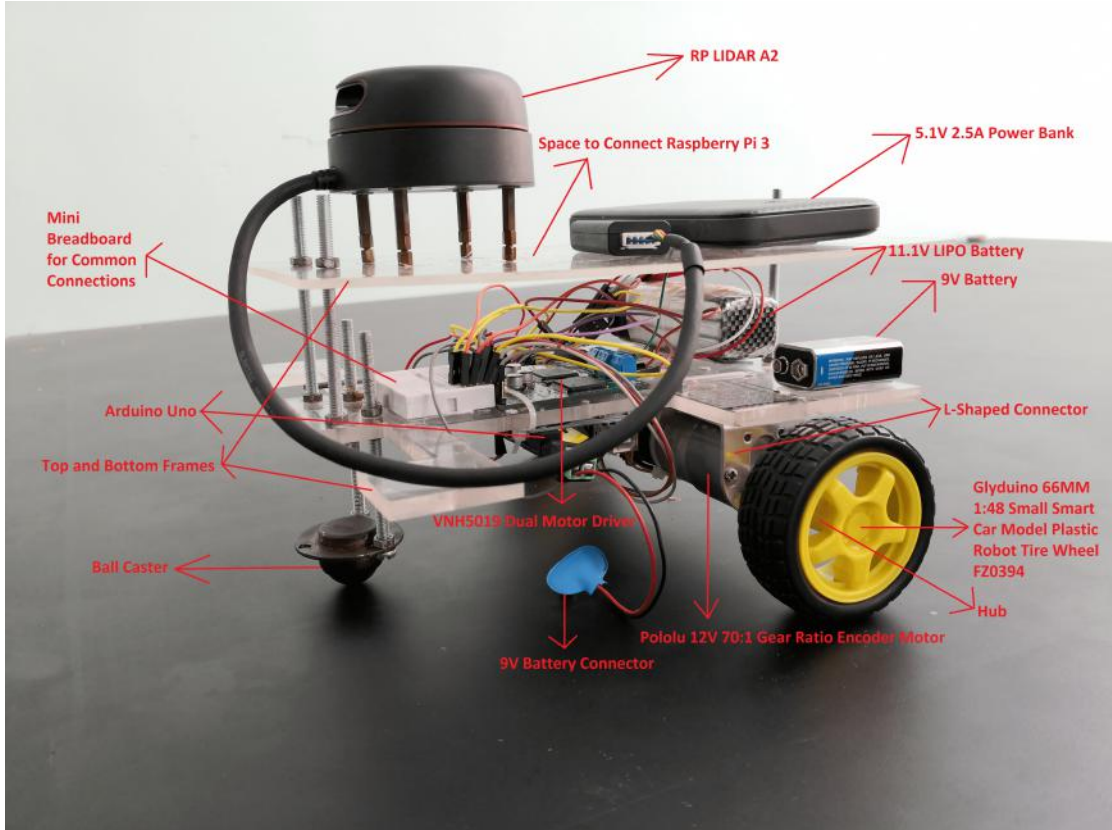


Figure 3.3: First Prototype

Our first robot prototype, shown in Figure 3.3, was composed of two levels. It had three wheels, with two of them connected to an encoder motor each, and one connected as a free rotating wheel/ball caster to keep the body in balance. Through our experience with testing, and via literature review, it was identified that the center of gravity in such a robot design has to lay inside the triangle formed by the wheels [14]. If too heavy of a mass is mounted to the side of the free rotating wheel, the robot tips over [14]. All of our major algorithm tests i.e. localization and mapping, PID/locomotion optimization, autonomous navigation, and obstacle detection and avoidance were performed on this prototype. However, due to the fact that this design was not suitable for the integration

of the payload delivery system, with the facial recognition and lock mechanisms, it was decided to come up with a new design, and integrate all of the prior work in it during the final stages of the capstone project.

#### 3.3.4 Final Prototype

Using the Google Sketchup Pro software, the final robot prototype design proposed in capstone 1 is shown in Figure 3.4.

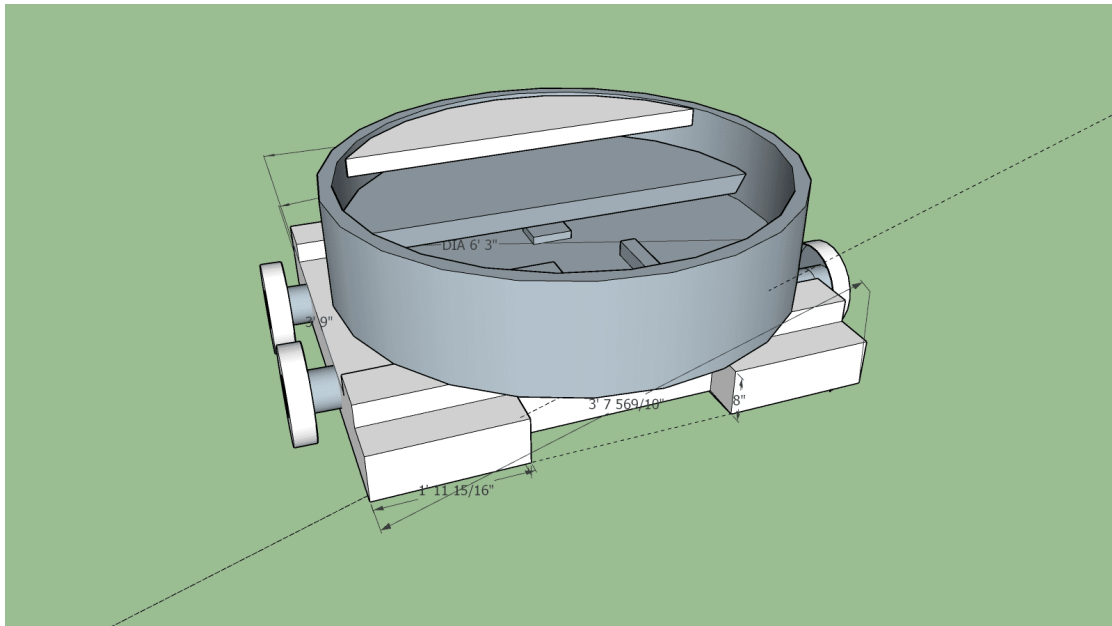


Figure 3.4: Final Prototype First Variant

Due to the complexity of the proposed design, and the unavailability of the 3D printing facility, this design was abandoned.

The re-imagined final robot prototype is composed of two driving wheels, connected to an encoder motor each, along with two ball casters, one at the front and the other at the back, to keep the body in balance. This design is more stable than the three wheel version because the center of gravity is to remain inside the quadrilateral formed by the four wheels, instead of a triangle. The design makes use of two different frames, rectangular in shape, and connected to each other via metallic spacers. The width of the robot chassis is kept 45 cm whereas, the length is kept 35 cm. According to the literature, such a design allows the front wheels to give negligible resistance to the

overall circular motion generated by the rear wheels when the robot takes a zero-radius 360 degree turn (pivot turning) [17]. Therefore, the torque applied by motors on each wheel gets harnessed properly, this results in a better performance without any drag, or wheel slip while taking a turn [17].

The bottom and top frames, designed on the Corel Draw software are shown in Figure 3.5, and Figure 3.6 respectively.

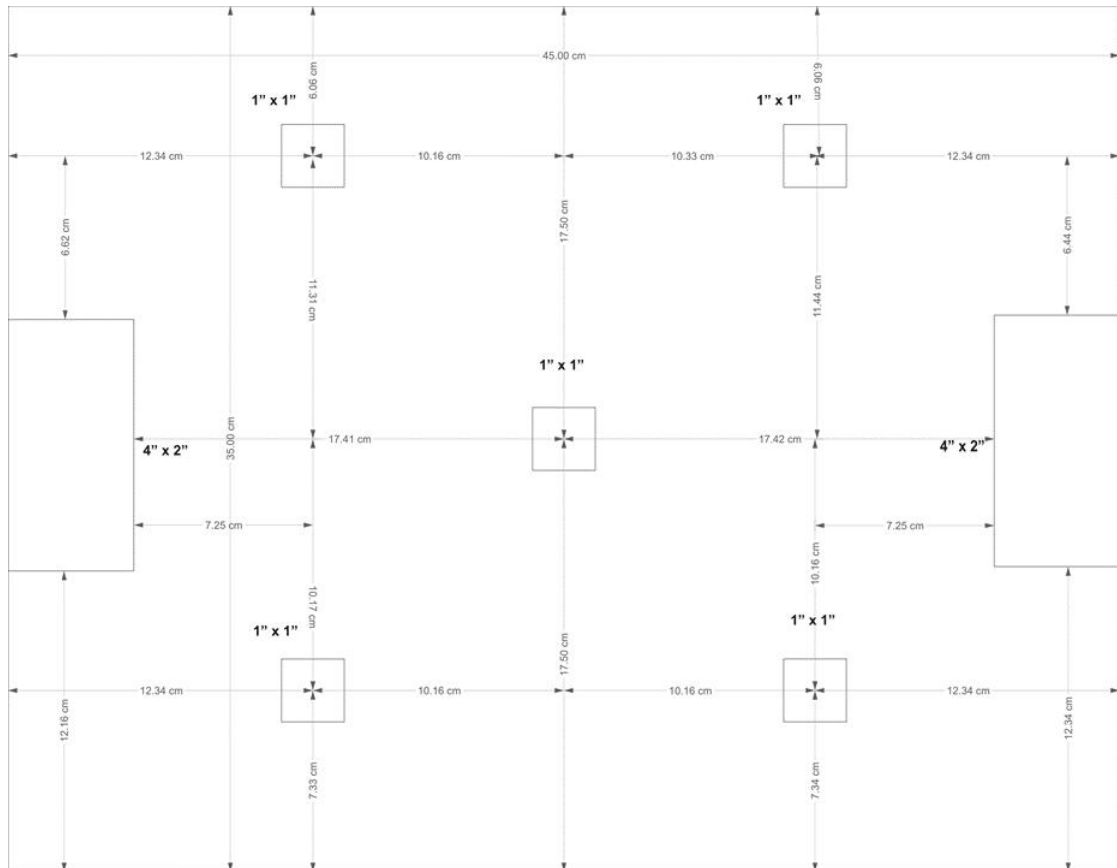


Figure 3.5: Bottom Frame

The two rectangles, one on the left, and the other on the right, are made for the installation of the wheels connected to the hubs, the encoder motors, and the L-shaped connectors.

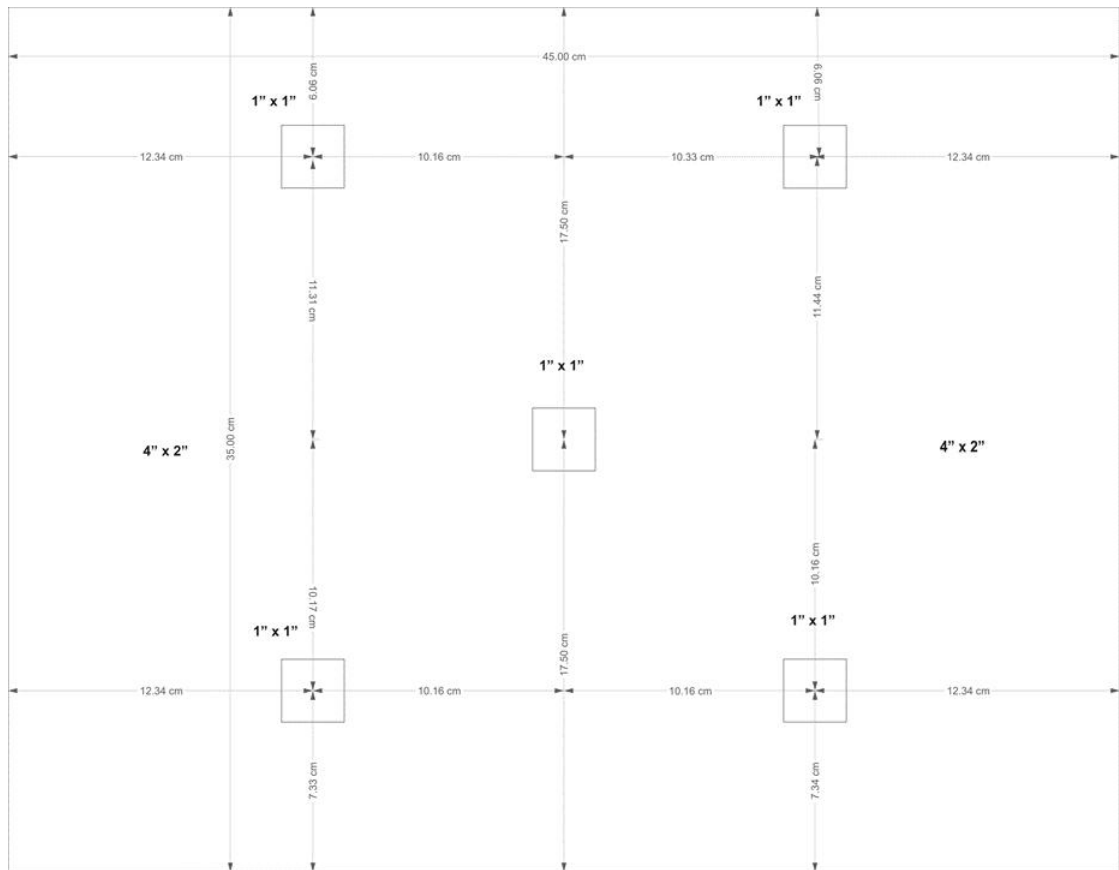


Figure 3.6: Top Frame

The five squares in the design are made to allow the wires to pass easily across all structural levels.

The Corel Draw frame documents, .cdr type, are used for the laser cutting procedure performed on two separate Acrylic sheets.

The final robot prototype structure is shown in Figures 3.7, 3.8, and 3.9 respectively.



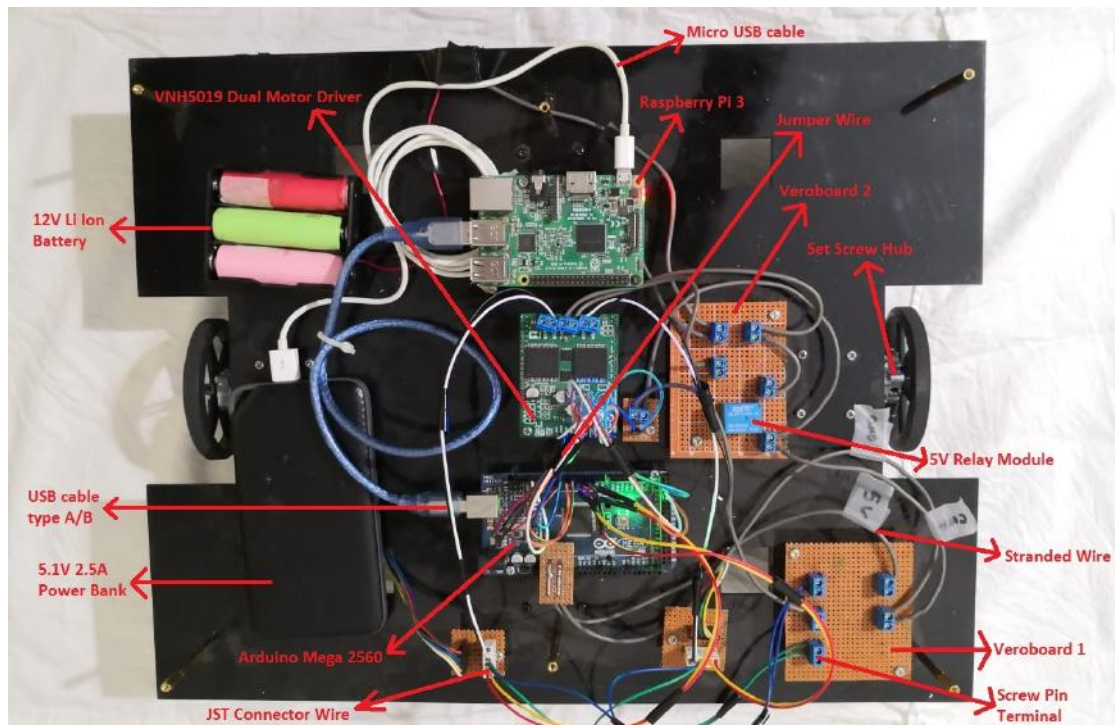


Figure 3.7: Internal View

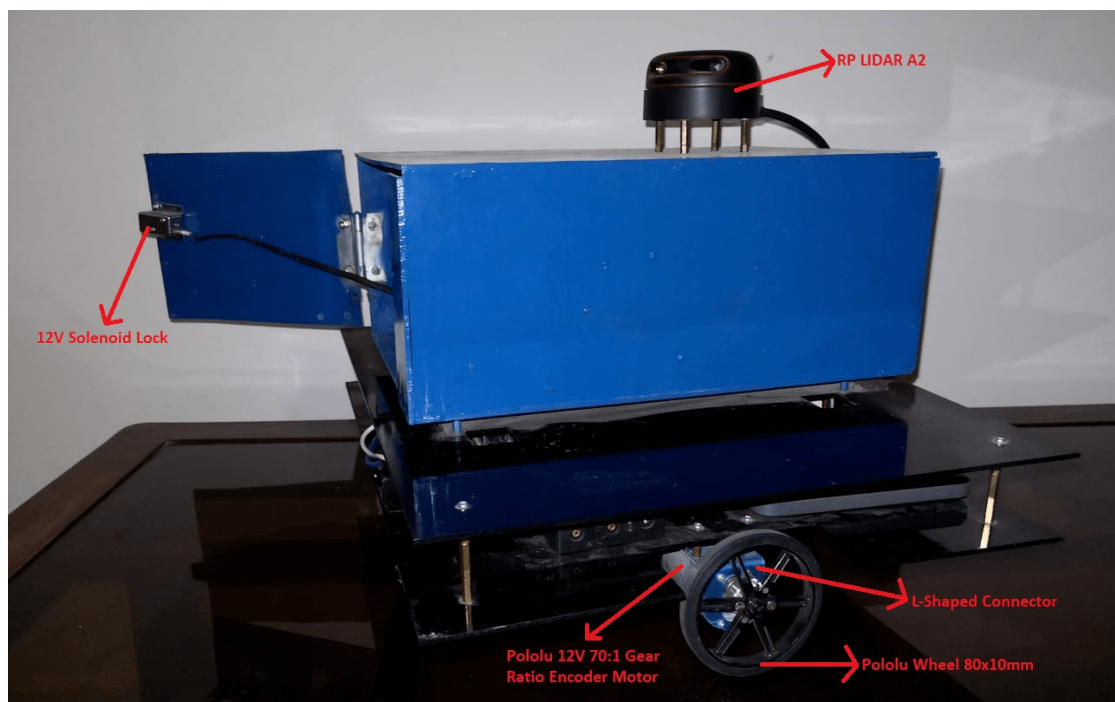


Figure 3.8: Side View

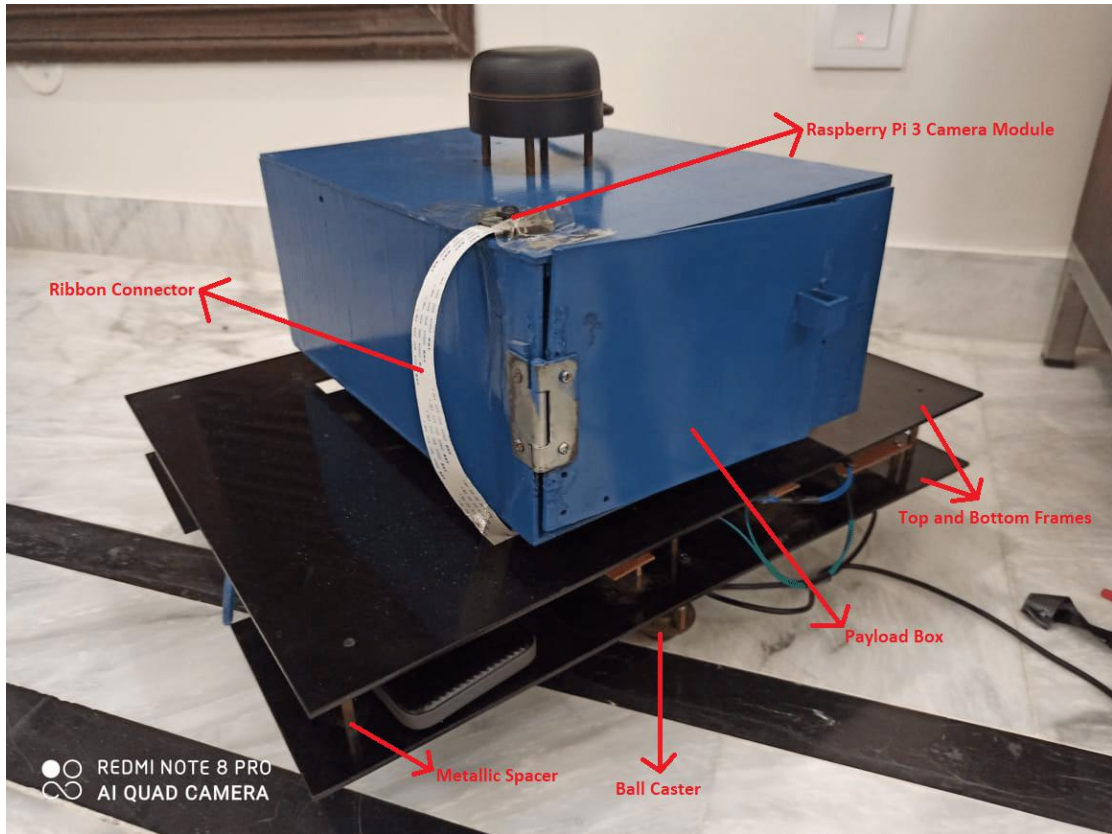


Figure 3.9: Elevated View

Several small and large sized metallic spacers are used for the installation of circuit and structural components. This ensured neatness in execution, and also made troubleshooting relatively easier to perform. Acrylic sheets are used for the frames, and the payload box because they are light in weight and durable. Ball casters are used because they are more stable as compared to wheel casters. Pololu wheels are used as they have a better grip on smooth or tiled surfaces. The other wheels considered were better suited to grassy and rough surfaces. Hubs are used to mount the wheels to the encoder motor shafts. The L-shaped connectors are used to grip the encoder motors.

### 3.3.5 Structural Components

The list of structural components used for the development of the final robot prototype include:

1. Two ball casters
2. Two L-shaped connectors

3. Two set screw hubs - 6 mm bore
4. Two Pololu wheels  $80 \times 10$  mm
5. Two Acrylic frames (bottom and top)
6. Small and large metallic spacers and screws
7. Acrylic sheet for the payload box
8. Veroboard, screw pin terminals, and header pins for the circuitry

### 3.4 Electrical Architecture

#### 3.4.1 Electrical System Block Diagram

The electrical system forms an integral part of the project. After the construction of the mechanical structure, an electrical system was required to allow the essential ROS algorithms to be tested.

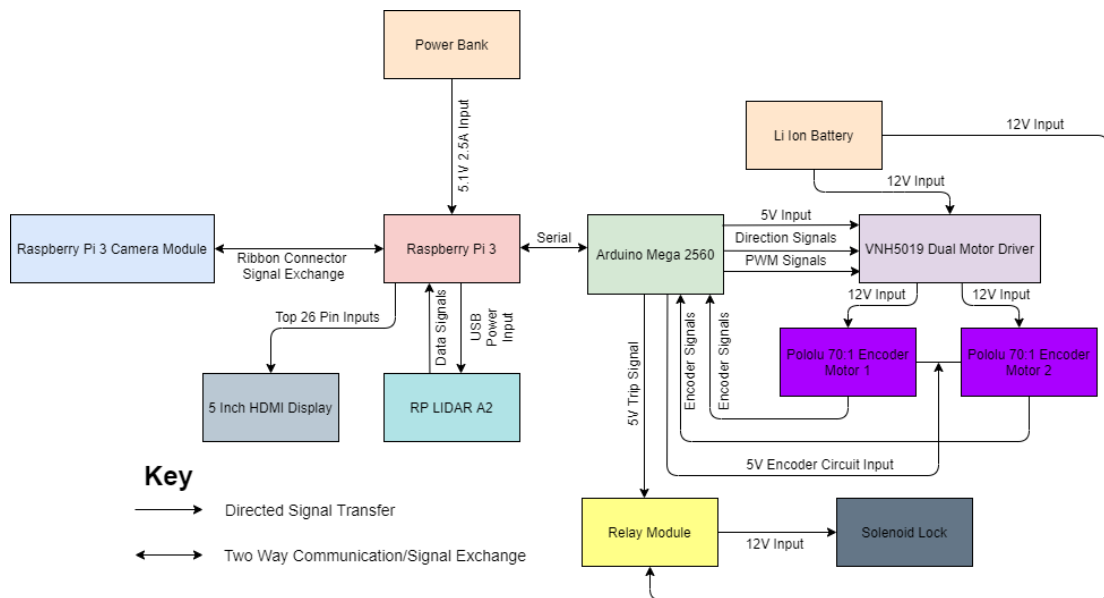


Figure 3.10: Electrical System Block Diagram

Figure 3.10 illustrates the electrical components used in the project, and their interconnections.

The final robot prototype wiring is illustrated in Figure 3.11, and Figure 3.12 respectively.

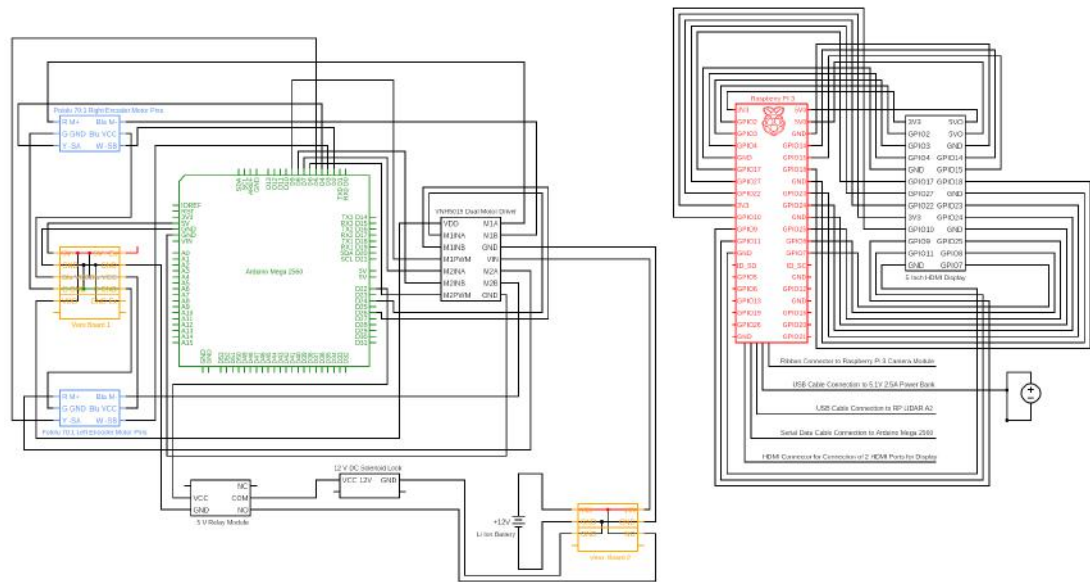


Figure 3.11: Overall Wiring

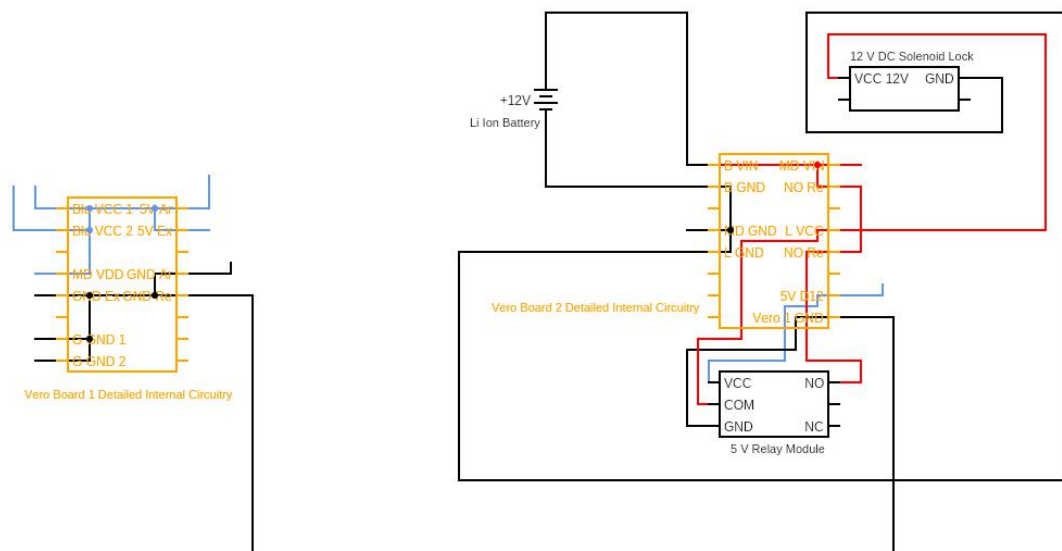


Figure 3.12: Wiring Associated with Veroboard 1 and Veroboard 2

To overcome the issue of loose connections between circuit components, and to account for the common potentials, Veroboards 1 and 2 are designed to make the overall

circuitry efficient. In order to establish the separability of the power and signal wires, for easier debugging, and visual identification, thicker stranded wires are used for the power connections, and thinner JST connector wires are used to accommodate the signals. Thus, the wires for the encoder signals, and the associated encoder circuitry are catered by these thinner identifiable signal wires whereas, the rest of the connections, such as, those to power the relay, and the motor are served by the thicker stranded wires. To facilitate the connections between the Arduino, and the motor driver for control signalling, jumper wires are used because they fit to the pins on each unit more easily.

### **3.5 Electrical Components (Working, Applications, and Specifications)**

#### **3.5.1 RP LIDAR A2**

The RP LIDAR A2 is used to detect the obstacles in the robot's vicinity. The LIDAR sensor rotates at a high speed, and transmits laser beams that rebound upon striking an obstacle. These laser beam values are sent to ROS where they are filtered, and accommodated in the respective libraries that are being used. The RP LIDAR library acquires the sensor data whereas, the Navigation Stack library acquires the obstacle detection data. With the use of a GUI called Rviz, it is possible to check where the obstacle is. Also, via other libraries, it is possible to estimate the depth of an obstacle as well. This, as a consequence, renders a lot of help with the obstacle detection and navigation part of the project. Due to the budget constraints, and the fact that this LIDAR model was available via the university for free, it is used instead of some other model.

Table 3.1 shows the essential specifications of the RP LIDAR A2.

Table 3.1: RP LIDAR A2 Specifications [18]

<b>Sample Frequency</b>	2-4 kHz
<b>Scan Rate</b>	5-15 Hz
<b>Distance Range</b>	0.15-6 m
<b>Angular Resolution</b>	0.45-1.35 °
<b>Laser Wavelength</b>	775-795 nm
<b>Laser Power</b>	3-5 mW

### 3.5.2 Pololu 70:1 Gear Ratio Encoder Motors

The Pololu 70:1 gear ratio encoder motors are attached to the wheels of the robot, they are connected to the Arduino via the motor driver. Encoder motors help obtain the position, as they convert the rotations of the shaft inside the motor into digital signals, which can then be fed to a microcontroller, such as, an Arduino to acquire the exact position of the robot. Each motor contains six wires, out of which four are used to power up and assign the direction of rotation to the motors, while the other two are used to transmit the position signals to the microcontroller. This motor type was less costly, and easily available via the university, which is why it is used.

Table 3.2 shows the essential specifications of the Pololu 70:1 gear ratio encoder motor.

Table 3.2: Pololu 70:1 Gear Ratio Encoder Motor Specifications [19]

<b>Manufacturer's Name</b>	Pololu
<b>Rated Voltage</b>	12 V
<b>Stall current</b>	5.5 A
<b>No-Load Current</b>	0.2 A
<b>No-Load Speed</b>	150 RPM
<b>Extrapolated Stall Torque</b>	27 kg.cm
<b>Extrapolated Stall Torque</b>	380 oz.in
<b>Maximum Power</b>	10 W

The unique feature about these motors is that they possess quadrature encoder, or incremental rotary encoder circuits that measure their speeds and directions. Each quadrature encoder is composed of two channels, which output pulse trains corresponding to



the rotation of a magnetic disk on a rear protrusion of the motor shaft, on the basis of the Hall effect. The number of pulses measured in a unit time are used to determine the speed of the rotating shaft of the motor. Since, the encoders work in such a way as to produce two pulse trains which are out of phase by 90 degrees, the leading/lagging behavior of the pulse trains with respect to each other is used to determine the direction of rotation. Figure 3.13 encapsulates this description in a visual form.

Refer to Appendix F.1 for the code of RPM Measurement.

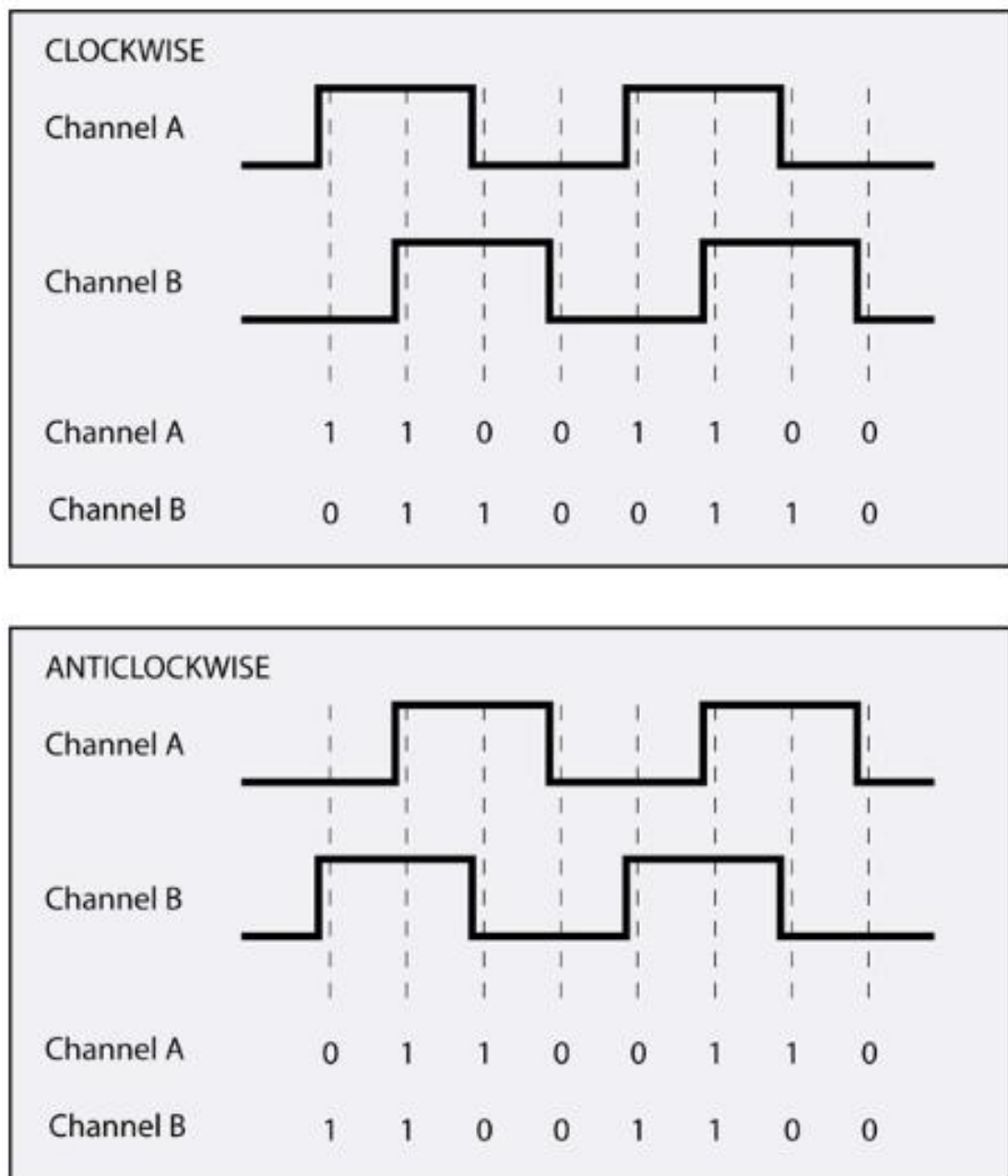


Figure 3.13: Working of a Quadrature Encoder

### 3.5.3 VNH5019 Dual Motor Driver

VNH5019 Dual Motor Driver is used on account of its protection abilities, as well as, its operation as a standard motor driver. It is highly robust, as it offers reverse-voltage protection up to -16 V, undervoltage and overvoltage shutdown, high-side and low-side thermal shutdown, short-to-ground and short-to-VCC protection, and can survive input voltages up to 41 V [20]. Another reason to use this motor driver is its compatibility with the Raspberry Pi, which is already a part of the project. A motor driver acts as an interface between the motors, and the control circuit, such as, an Arduino. A motor requires a high amount of current whereas, the controller circuit operates on low current signals. Thus, the function of a motor driver is to take a low current control signal, and convert it into a higher current signal that is capable to drive a motor.

Table 3.3 shows the essential specifications of the VNH5019 Dual Motor Driver.

Table 3.3: VNH5019 Dual Motor Driver Specifications [20]

<b>Manufacturer's Name</b>	Pololu
<b>Minimum Operating Voltage</b>	5.5 V
<b>Maximum Operating Voltage</b>	24 V
<b>Continuous Output Current Per Channel</b>	12 A
<b>Peak Output Current Per Channel</b>	30 A
<b>Current Sense</b>	0.14 V/A typ
<b>Maximum PWM Frequency</b>	20 kHz

### 3.5.4 Solenoid Lock with Relay Module

A 12 V Solenoid lock coupled with a 5 V relay is used to operate the payload box. A relay controls one electrical circuit by opening and closing the contacts in another circuit. When a relay contact is Normally Open (NO), there is an open contact when the relay is not energized. When a relay contact is Normally Closed (NC), there is a closed contact when the relay is not energized. In either case, applying an electrical current to the contacts changes their state. The relay module gets energized via a 5 V trigger signal from the Arduino, the signal is transmitted on the basis of the algorithms running on the Raspberry Pi. Since, the relay is set in the NO state, as soon as it is



energized, the contact closes, and a 12 V signal is transmitted to the Solenoid lock via the COM port. This signal produces a current that travels through a coil of wire in the lock to create a magnetic field that attracts a metal plunger or locking pin, and it moves to an unlocked position against a spring. Thus, the door gets unlocked. As soon as the trigger signal is removed, the relay, and consequently the Solenoid lock are both de-energized, and the door gets locked once again. Both the relay module and the Solenoid lock are connected to each other via Veroboard 2. A relay is used because there is a need to control the lock, which requires a high voltage signal, with a 5 V Arduino output. The Solenoid lock and relay models used are the popular, regular, and easily available ones, those that match our power capacity.

#### 3.5.5 Lithium Ion Battery

A collection of three rechargeable Lithium Ion cells are used to power the motor driver, which supplies power to the motors, and these cells are also used to operate the lock. Each cell is rated at 4 V to provide a maximum total of 12 V. The option of using a Lithium Polymer battery (LiPo) was also available, the rechargeable LiPo battery was composed of three cells, with each cell rated at 3.7 V with a capacity of 2200 mAh to provide a maximum total of 11.1 V at 6600 mAh. Due to the lower voltage and capacity ratings of comparatively priced LiPo battery models, as well as, the need to purchase additional battery charging circuitry for a LiPo battery, the Li Ion battery proved to be a more economical choice. The Lithium Ion cells are encased in a battery holder from which the positive and negative terminals of the battery are connected to Veroboard 2 to supply power to where it is needed.

#### 3.5.6 Power Bank

A power bank is used to supply power to the main controller i.e. the Raspberry Pi. The unit is a Huawei Technologies model AP08Q, rated at 10,000 mAh (38.0 Wh). The main reason to use this power bank is the availability of a 5.1V/2.5A USB output port that matches the power requirements of the Raspberry Pi model being used. With the

power requirements being sufficiently met, there was no reason to design, and construct the additional power electronics circuitry to convert the 12 V input to an output to match the Raspberry Pi's power requirements. Using a serial connection, via the USB cable type A/B, the power supplied to the Raspberry Pi is also fed into the Arduino and hence, the power needs of both systems are satisfied.

### 3.5.7 Raspberry Pi 3 Camera Module

A Raspberry Pi 3 CMOS camera module is interfaced with the Raspberry Pi via the CSI camera port. On the basis of the commands from the Raspberry Pi, the camera captures an image, and passes it on to the Python server, where the facial recognition function is being implemented. The Python server validates the face of the customer, and returns the result. On the basis of that result, the Raspberry Pi sends a command to the Arduino to open the lock. Due to the budget constraints, and the fact that this camera type was available via the university, it is used instead of some other model.

Table 3.4 shows the essential specifications of the Raspberry Pi 3 camera module.

Table 3.4: Raspberry Pi 3 Camera Module Specifications [21]

<b>Module Type</b>	V 2.1
<b>Sensor</b>	Sony IMX219PQ
<b>Still Resolution</b>	8 Megapixels
<b>Sensor Resolution</b>	3280 x 2464 Pixels
<b>Video Modes</b>	1080p30, 720p60 and 640x480p90
<b>Size</b>	25 mm x 23 mm x 9 mm
<b>Weight</b>	> 3 g
<b>Connection to Raspberry Pi</b>	Ribbon Cable
<b>Optical Size</b>	1/4"

### 3.5.8 Raspberry Pi 3

The brain of the entire project is the Raspberry Pi model 3B single board computer. This unit includes the essential features for the project, which are:

1. HDMI output for connectivity to the display.

2. Wireless LAN capability to communicate with the servers, and the admin computer used in the project.
3. USB ports for interfacing with the LIDAR and Arduino.
4. Quad Core 64 bit CPU with 1 GB of RAM to handle the computational tasks including mapping, and path planning algorithms, performing high and low level communication (running ROS commands, and communicating the serial instructions, and sensor readings with the Arduino), and handling peripherals, such as, the LIDAR sensor.
5. Extended GPIO pins (numbered at 40) to interface with the necessary peripherals including an HDMI display.
6. CSI camera port for connecting a Raspberry Pi camera to capture the input for the facial recognition feature.

To fulfil the above requirements, the model 3B was preferred over other Raspberry Pi models. The model serves as the central computer to host the ROS code, and manages all the algorithms being executed, from gmapping for 2D environmental mapping, to the A Star algorithm for path planning. The Raspberry Pi also handles the communication with the LIDAR, to transmit and receive the laser pulses, via the LIDAR package set up in ROS. The unit is also responsible for the serial communication with the Arduino, mainly by setting up a publish/subscribe communication relationship with the Arduino. Through this link, the Raspberry Pi receives information, such as, odometry readings from the Arduino and transmits the PWM values to it to facilitate the desired locomotion of the motors, in accordance with the higher level algorithms running on the Raspberry Pi. The unit is also interfaced with a Raspberry Pi CMOS camera to trigger the camera's ability to capture images. Furthermore, it subsequently passes the image on to the facial recognition function implemented on a server. The lock mechanism for operating the payload box is also partially handled by the Raspberry Pi via ROS.

Table 3.5 shows the essential specifications of the Raspberry Pi 3.

Table 3.5: Raspberry Pi 3 Specifications [22]

<b>Model</b>	3B
<b>SoC</b>	Broadcom BCM2837
<b>CPU</b>	4x ARM Cortex-A53 at 1.2 GHz
<b>GPU</b>	Broadcom VideoCore IV
<b>RAM</b>	1 GB LPDDR2 (900 MHz)
<b>Bluetooth</b>	Bluetooth 4.1 Classic, Bluetooth Low Energy
<b>GPIO</b>	40-pin header, populated
<b>Ports</b>	HDMI, 3.5 mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

### 3.5.9 Arduino Mega 2560

An Arduino Mega 2560 communicates with the Raspberry Pi serially, via a publish and subscribe architecture. The Arduino is responsible to send the odometry readings to the Raspberry Pi, via the encoder motors, and receive the PWM values from the Pi to support the desired motor locomotion. These PWM values are sent to the motor driver. Also, to control the direction of movement, the Arduino sends the direction values to the motor driver, based on the algorithms running on the Raspberry Pi. The Arduino also provides the logic power supply to the motor driver to power the internal pull-ups on the ENA and ENB enable lines. Moreover, to power the encoder circuits in the encoder motors, the Arduino provides the logic power supply to the encoder motors as well. Furthermore, the Arduino also sends a 5 V trip signal to the relay module in order to operate the lock for the payload box, on the basis of the algorithms running on the Raspberry Pi.

The Arduino Mega 2560 has 256 kB of Flash memory, giving it 8x more memory space in comparison to an Arduino Uno and Micro, that have 32 kB [23]. The Mega 2560 has the most SRAM (Static Random-Access Memory) space with 8 kB, which is 4x more than the Uno, and 3.2x more than the Micro [23]. With more SRAM space, the Arduino has more space to create and manipulate variables when it runs. Another reason to use Mega 2560 was the availability of a large number of digital I/O, digital I/O with PWM, and analog pins.

Table 3.6 shows the essential specifications of the Arduino Mega 2560.

Table 3.6: Arduino Mega 2560 Specifications [23]

<b>Processor</b>	ATmega2560
<b>Clock Speed</b>	16 MHz
<b>Flash memory</b>	256 kB
<b>EEPROM</b>	4 kB
<b>SRAM</b>	8 kB
<b>Voltage Level</b>	5 V
<b>Digital I/O Pins</b>	54
<b>Digital I/O with PWM Pins</b>	15
<b>Analog Pins</b>	16
<b>USB Connectivity</b>	Standard A/B USB
<b>Shield Compatibility</b>	Yes
<b>Ethernet/Wi-Fi/Bluetooth</b>	No, a shield/module can enable it

(Note: The 5 Inch HDMI Display is not used in the final robot prototype, as the touch interface of the display provided did not function adequately).

## 3.6 ROS

### 3.6.1 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open source framework for use in the robotics industry. The framework works on the Linux architecture, and offers features, such as, low-level (hardware) device control, sensor fusion, robot simulation through Gazebo etc.

### 3.6.2 Why ROS?

ROS has a large community support. The regularly updated, and largely available libraries in ROS offer convenience to new users. Since, the project relies on the fundamentals of localization and mapping, ROS is used. Furthermore, the project also relies on the routing of data between the server, and the low-level hardware (Arduino). Therefore, to deal with such issues, an embedded system, such as, a Raspberry Pi is used.

The team decided to load the Lubuntu OS on the Raspberry Pi since, it allows multi-threading of processes to be done, allowing it to employ less memory. Further, the team also loaded the Lubuntu OS packaged with the ROS Framework, and setup a Catkin work space. After the creation of the work space, the team downloaded the libraries that were essential in developing the software base for the project.

The fundamental reasons to use ROS are as follows:

1. ROS is a famous open source framework that includes packages for a wide range of applications, and contains pre-built packages for a majority of the actuators, sensors, and other hardware, as well as, other packages to build drivers for the components which do not have drivers.
2. Complex projects require suitable simulations before execution. ROS comes pre-bundled with Rviz and Gazebo modules through which a completely modifiable virtual environment can be setup for necessary testing of the robot's functions. Moreover, these simulations can also benefit in run time. Rviz can also be used to display a live map for SLAM.
3. ROS offers an extensive language compatibility. This trait can turn out to be quite useful when large teams having members with different programming backgrounds come together to build something. A system based on ROS can include packages and libraries from different programming languages, which as a whole work together. It is also significant when a cross platform system is built, or a vast range of packages is required to be incorporated which might be written in a multiple languages.

### 3.6.3 ROS Concepts

There are certain characteristics and paradigms of ROS which differentiate it as a software suite. These include concepts, such as, nodes, topics, publishers, ad subscribers, messages, and launch files, which are highlighted henceforth:

1. Message - Messages are data structures with particular fields that specify the type of data for example, String, Boolean, Integer etc. that is being transferred between nodes.
2. Node - A ROS node can be described as a process performing a computation. It is simply an executable program that runs as a part of the application created by the developer. Nodes are typically combined into a graph, and these nodes communicate with each other through protocols including topics, services, and actions. The purpose of handling computations through the encapsulations, called nodes, includes reducing the complexity of code by separating an application into more manageable chunks, which also helps in scaling it up and make it more modular. Nodes also improve the application's fault tolerance since, they communicate via ROS without being directly linked together so that if one node crashes, the others are not affected.
3. Topic - Information exchange between nodes in ROS occurs via buses which are known as topics. They determine what type of data can be transmitted between nodes. Topics establish a decoupling between the production and use of data. Nodes are unaware of which node they are communicating with. Rather, they publish to a particular topic when they want to send data, and they receive data by subscribing to the relevant topic.
4. Publisher and subscriber - The publisher/subscriber (pub/sub) mechanism is the paradigm by which ROS operates. This mechanism facilitates the exchange of data between nodes by subscribing to topics to receive data, and publishing to topics to send data. The pub/sub communication model also facilitates many to many communication between multiple publishers and subscribers which are transmitting and receiving nodes, connected to each other by topics.
5. Launch file - These files provide convenient means to launch multiple nodes, and initialise parameters in an efficient and manageable manner. They usually contain code where the parameters for the application, such as, PID settings for the robot,

are stored, as well as, code to initialise ROS packages.

### 3.6.4 ROS Setup

The project is built on ROS Kinetic. While there are later versions of ROS currently available to the public, ROS Kinetic is the stable one having a Long Term Stable (LTS) version available. It takes a couple of months for packages to migrate from previous distributions to the new ones, making LTS a better option than latest unstable release.

#### 1. Desktop/Laptop Setup

ROS, originally made for Linux environments is still in the process of transitioning its full compatibility to Windows. Therefore, for full functionalities and stable operation, Debian based environments, such as, Ubuntu is preferred. The team has used Ubuntu 16.04 LTS (Xenial Xerus), compatible with ROS Kinetic. For installation on desktop/laptop machine, the standard ROS installation instructions are available on the ROS Wiki site [24].

#### 2. Raspberry Pi Setup

The project consists of multiple stationary and mobile nodes. Instead of installing Ubuntu on Raspberry Pi, the use of Lubuntu was preferred. Lubuntu (Light Ubuntu) is a lighter and faster variant of Ubuntu specifically designed for computers with limited processing power, such as, the Raspberry Pi. A Lubuntu and ROS compiled image is provided by Ubiquity Robotics. The image, and the complete tutorial on how to install ROS and Lubuntu together can be found on the Ubiquity Robotics site [25]. The image can be directly burned in an SD card, bypassing any need for the installation that was done earlier for the desktop/laptop.

#### 3. Catkin Work Space Setup

Catkin work space is the build system for ROS. In ROS, targets can be the libraries, packages, or a dynamically created script which is not said to be hard coded. A build system handles the tasks of generating targets from all program



scripts. These targets can be used by the end user, without worrying about the underlying processes, or which libraries are used behind the scenes. More details about the Catkin work space can be found on the ROS Wiki site [26].

Once ROS has been installed, the Catkin work space needs to be built and initialized for the first use. The tutorial to setup, and initialize the Catkin work space can be found on the ROS Wiki site [27].

The complete tutorial for ROS can be found in Appendix B.

### 3.6.5 ROS Packages Used

This section briefly touches upon the packages used in the project. All open source, third-party packages can be installed into ROS by cloning their github repositories to the Catkin work space `./src` folder, and rebuilding the workspace using the *catkin make* command in the CLI.

#### 1. Rosserial

The Rosserial package is used to establish communication between the Raspberry Pi and the Arduino. The data that is being transferred over to the Arduino includes the PWM values for the motors, and the lock mechanism commands. The Arduino uses the package to subscribe to the ROS topics, and is able to publish the encoder values back to the Pi. The package uses the standard Universal Asynchronous Receiver Transceiver (UART) connection with 115200 baud rate. More details on this package can be found on the ROS Wiki site [28].

#### 2. rplidar\_ros

The rplidar\_ros package is used to channel the RP LIDAR sensor to ROS. The package publishes the scan data on the `/scan` topic on ROS. More details related to this package can be found on the ROS Wiki site [29].

#### 3. Navigation Stack

The Navigation Stack is used to navigate the robot control using odometry, and a pre-generated map that was created using the Cartographer library. Rviz is

used to set the current location and pose of the Robot on the map, and to set the goal position of the robot that was desired for it to travel to. The Navigation Stack uses the `amcl` node for localization, the differential drive node to send its speed and direction, and lastly the map server node to subscribe to the map. The package translates these commands into a trajectory, and transmits it to the robot using the `cmd_vel` topic. Rviz will not be used for the final demonstration as the whole process is being coded together into a file by the name `MainLoop` (found in Appendix G.2). More details related to this package can be found on the ROS Wiki site [30].

#### 4. Gmapping

Gmapping is a ROS package that provides the OpenSlam Gmapping algorithm which constructs a 2D occupancy grid map of the environment, utilizing laser data and odometry information. More information related to the package can be found on the ROS Wiki site [31]. The algorithm can be found on the Openslam site [32].

Gmapping is a SLAM algorithm which builds a map using odometry and sensor data. It is highly effective when access to reliable odometry is available, and it performs well even in symmetric environments with very less dynamic obstacle variation.

#### 5. tf

In order to re-use software components, and better integrate them, a shared convention for coordinate frames is required by the developers of libraries, drivers, and models. Shared conventions render a specification for the developers that create drivers and models for mobile bases. The `tf` package allows the user to keep track of multiple coordinate frames over time. It keeps the relationship between the coordinate frames in a tree structure, buffered in time, and allows the user to transform vectors and points among various other things, between any two coordinate frames, at any time. More information on this can be found on the ROS

website [33]. Information related to the tf package can be found on the ROS Wiki site [34].

A visual representation of various coordinate frames can be seen in Figure 3.14

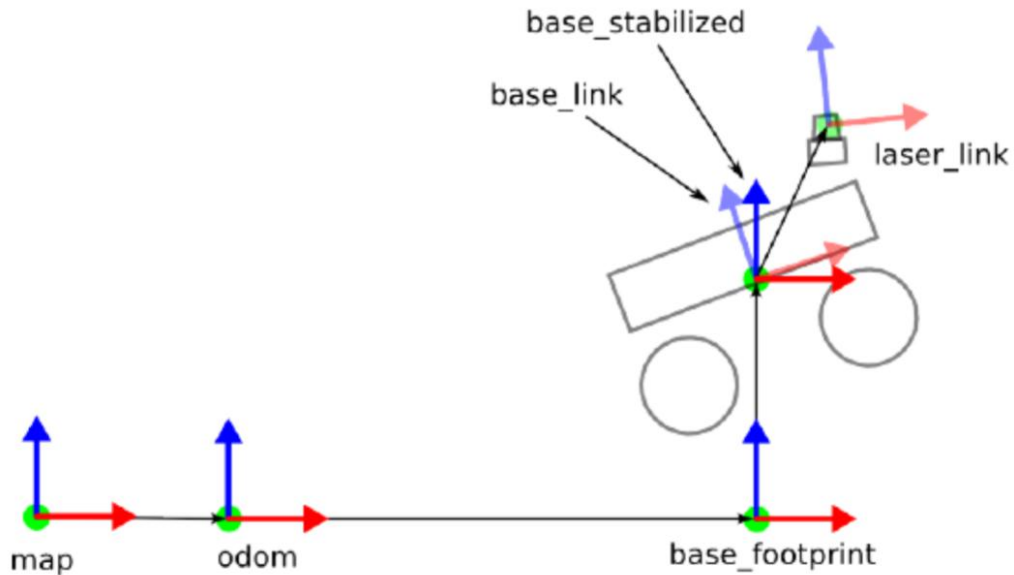


Figure 3.14: The tf Transformed Coordinate Frame for a Differential Drive Robot

tf manages the various transforms that are required to be published for a robot. There are standard terms defined for a robot's physical structure, such as, `base_frame`, `laser_frame` etc. The robot's frame of reference is the base frame, and it is also referred to as the base link. The laser frame is the coordinate frame of the LIDAR, it is placed on the top of the robot, and is called base laser. Continuous publishing of transformation from `base_link` to `laser_link` is done by the tf, denoting the position of the LIDAR with respect to the center of the robot. The package also maintains dynamic transforms, such as, the `odom` to `base link` transform, which represents the position of the robot base frame, `base link`, with respect to the starting position, which is the odometry. Hence, frame name `odom`. A visual representation of the system transform tree is shown in Figure 3.15.

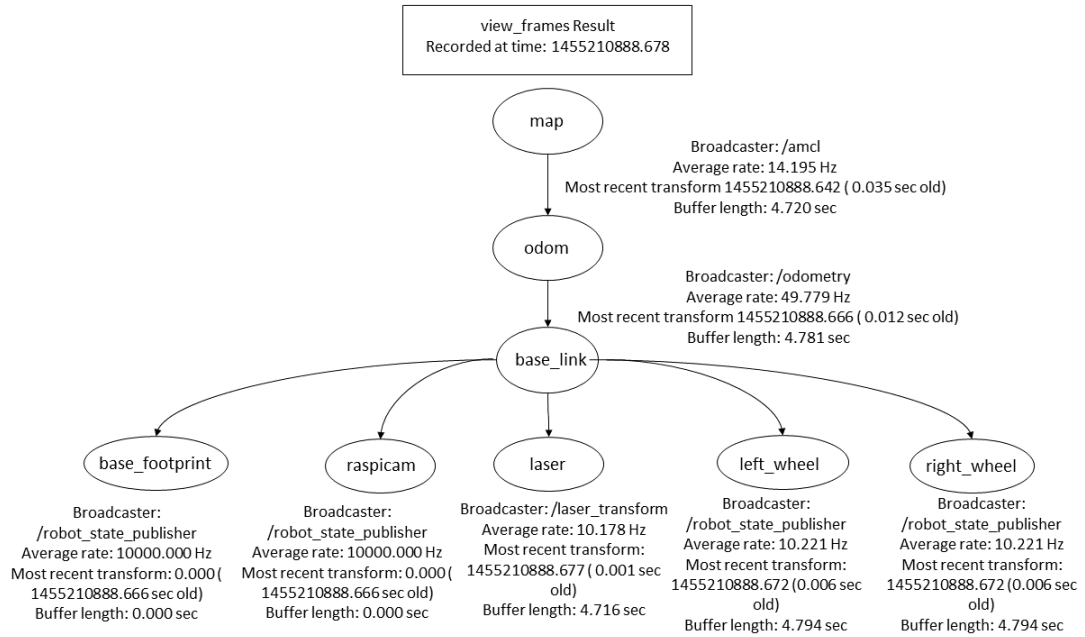


Figure 3.15: Visual Representation of System Transform Tree

With ROS, it is possible to publish both the static, and the dynamic transforms. An example of the static transform is the base link→laser link transform. The laser is rigidly fixed to the robot, hence its transform is static with respect to the robot. On the other hand, the odom→base link transform is dynamic since, it keeps changing as the robot moves, hence it is a dynamic transform. This dynamic transform changes are being published by the amcl node.

## 6. differential\_drive

The differential\_drive package provides tools required to interface a differential drive wheeled mobile robot to the ROS Navigation Stack. The purpose of this package is to create an implementation of a differential drive controller that is independent of the specific hardware, for instance, a particular microcontroller unit, used to implement the robot.

This package takes in messages from the Navigation Stack, and converts them into the left wheel and right wheel messages, which go on to serve as the voltage levels transferred to the motor driver in the form of PWM signals. The package also performs the task of receiving the encoder signals from the hardware, and

generates messages used in odometry and location referencing for the Navigation Stack, in the form of tf transform messages. To maintain the hardware abstraction, the speed messages are communicated in the form of x-axis metres/second and z-axis radians/second.

The package also provides a PID controller that uses feedback from the rotary encoders attached to the motors to form a closed loop control system to control the wheel velocities. Two PID controllers are instantiated, corresponding to each wheel. The package also provides a node for odometry calculations through the encoders of the motors. The ticks detected by each encoder are counted and published by the low level controller (Arduino) to the differential drive package, using the serial communication package, Rosserial. The odometry node subscribes to these messages, and calculates the odometry values of the robot.

## 7. Google Cartographer

The reasons for using Google Cartographer, instead of Hector SLAM and Gmapping are listed as follows:

- (a) Hector SLAM is an efficient algorithm used to make a map of the environment but it does not use odometry information. This can cause some problems while mapping, such as, it does not know where it currently is, and uses the LIDAR to make an assumption of its location. This makes the map rather unreliable, as it can cause certain map areas to be in those places that do not really align with real world environment.
- (b) Gmapping overcomes this area, and allows SLAM to be used however, it was not used because later it was found out that SLAM was not really needed. There was just the need to map the structure, and move the robot about it. Furthermore, some of its unreliabilities are that it makes a map in one go, and it does not make a map on top of it to make sure what it is doing is correct.

Figures 3.16 and 3.17 show Gmapping, Hector SLAM, and Google Cartographer side by side.

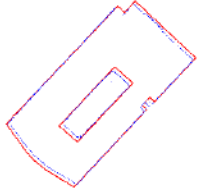
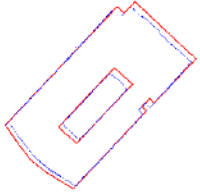
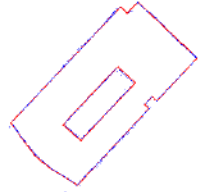
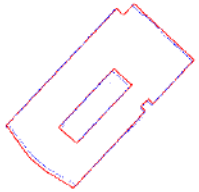
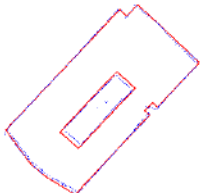
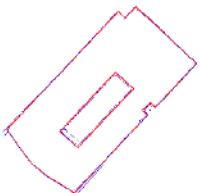
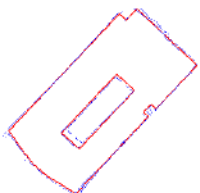
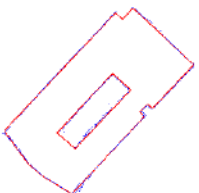


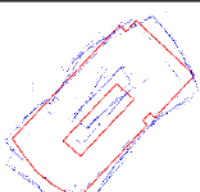
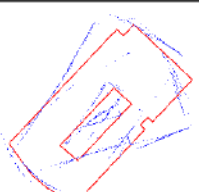
SLAM method	Slow	Fast/Smooth	Fast/Sharp	No loop closure
Gmapping				
Cartographer				
Hector SLAM				

Figure 3.16: An Overview of Gmapping, Hector SLAM, and Google Cartographer

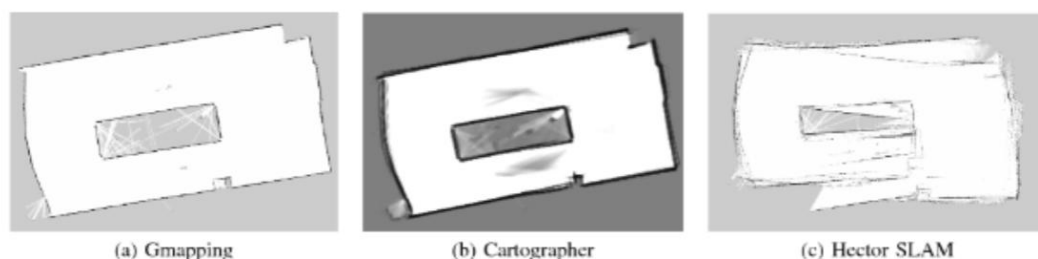


Figure 3.17: Mapping Results of Gmapping, Hector SLAM, and Google Cartographer

This is where the Cartographer comes in, Cartographer is a Google based algorithm that can be used for mapping, as well as, just SLAM. It can improve the map if it is in the area long enough, and can make a map on top of the other map, to make it more efficient. It also uses the odometry information that is provided to it by the Arduino encoders. This makes Cartographer one of the best mapping algorithms, and the optimal solution for the project's scope. More related

information can be found on the Google Cartographer site [35].

Figure 3.18 shows an overview of the Google Cartographer package.

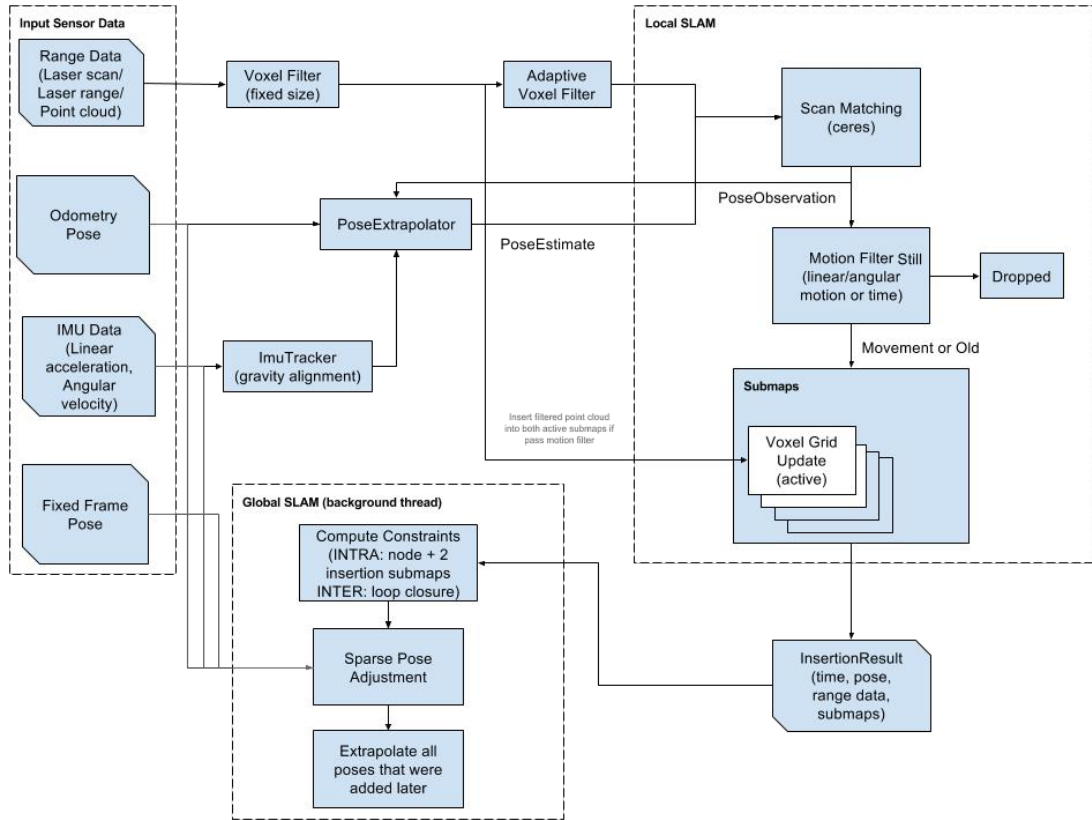


Figure 3.18: High Level Overview of Google Cartographer Package

All the necessary configuration and launch files associated with ROS and the final prototype can be found in Appendix H.

## 3.7 Facial Recognition

### 3.7.1 Purpose and Overview

The team wanted to create a solution to be used in indoor spaces, not only to deliver perishables or lunch supplies, but also to aid in interdepartmental communication, which includes the delivery of essential and confidential packages. It was necessary to make the working system safe and secure for the end user. Hence, it was decided to add a facial recognition module to offer an extended biometric security.

A facial recognition module can identify or verify a person from a digital image, or a

video frame from a video source. It is also known as a Biometric Artificial Intelligence based application that can uniquely identify a person, by analyzing the patterns based on the person's facial textures and shape.

### 3.7.2 Procedure

The facial recognition module is built on OpenCV, and the scikit-learn modules of Python. It was decided to go for OpenCV since, it is completely open source, and has a tremendous amount of community support. OpenCV comes with a trainer as well as detector. There are two program flows in the facial recognition module, training, and recognition. The training module receives a token ID, generated in the app against each user's entry. An input video is given along with the token to train the model. Once the model is successfully trained, it gives the response code 201. The recognition module is connected to the robot's on board Pi camera, the robot on completion of the booking prompts the Pi camera to capture an image, and once an image is captured, and sent to the server, the module recognizes, and outputs the token ID. If the token ID matches with the current receiver's token ID, then the lock API on the Arduino opens the lock successfully.

The block diagram of the facial recognition flow is given in Figure 3.19.



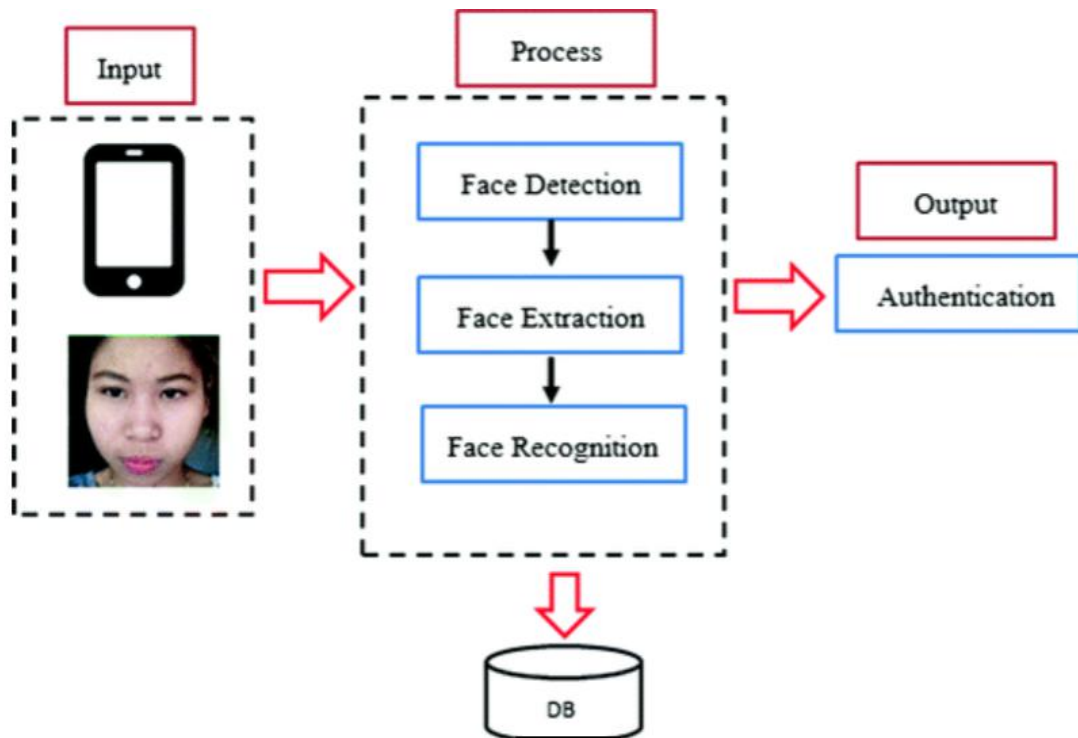


Figure 3.19: Block Diagram of Facial Recognition Flow

Face detection falls under a specific case of object-class detection. In object-class detection, the task is to find the locations and sizes of all objects in an image that belong to a given class. Real life examples of this include pedestrians, cars etc.

The facial recognition module can be broken down into two major components, detection and recognition as follows:

1. To apply face detection, which detects the presence and location of a face in an image, but does not identify it.
2. To extract the 128-d feature vectors, called embeddings, that quantify each face in an image

Now once OpenCV is used to detect a face from an input provided. The image can be processed further to find whether that image is a match with any of the existing entries or not. To do that, embeddings (128-d feature vectors) need to be extracted from the input.

The block diagram of the facial recognition module is shown in Figure 3.20.

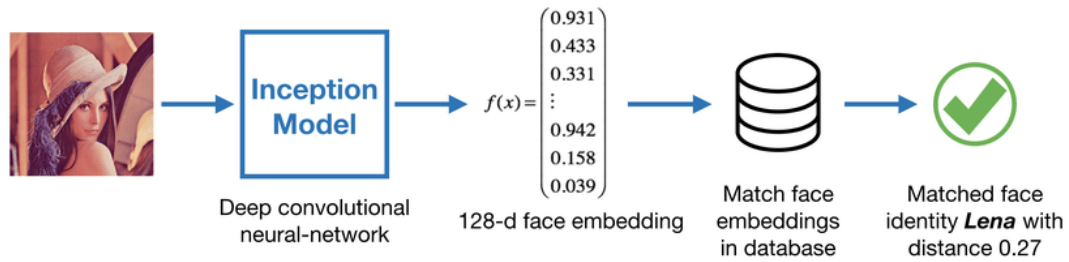


Figure 3.20: Block Diagram of Facial Recognition Module

The tutorial to setup the facial recognition module can be found in Appendix C.

### 3.8 Android Application

#### 3.8.1 About

This is the module that interacts directly with the user. The user creates an account via the app. The user can then use the app to send packages to other users through the app, by first calling the robot to his or her location, and then tagging the receiver. During the process, the user can view the live location of the robot. The user also has the liberty to view previous deliveries. This module directly communicates with two other modules, i.e, the central server, and the Python server. The app communicates with the central server to place bookings, and with the Python server to send the user's face video for training the model. This allows the user to open the container at the time of receiving the package, by using the facial recognition feature.

#### 3.8.2 Signup and Login Page

Figure 3.21, on the left, shows the signup page. The signup page is the first step towards creating an account. New users have to specify their email, name, and password in order to create their account.

Figure 3.22, on the right, shows the login page. Existing users can use their email, and password in order to login to the app. Login is required only once, so the next time the user opens the app, he or she does not have to login again.

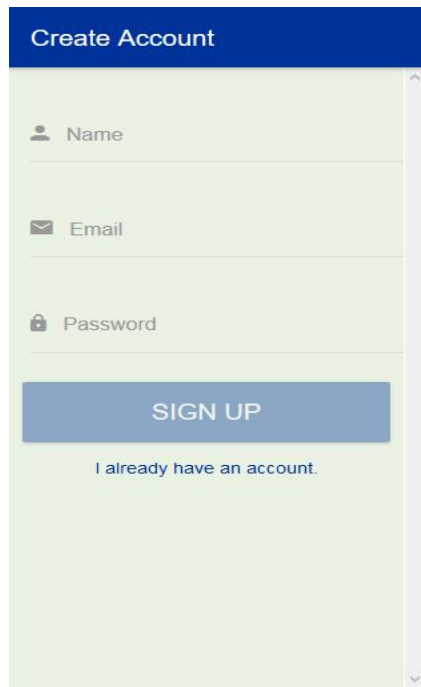
A mobile app interface for creating a new account. It features a blue header with the text "Create Account". Below the header, there are three input fields: "Name" with a person icon, "Email" with an envelope icon, and "Password" with a lock icon. At the bottom, there is a blue "SIGN UP" button and a link that says "I already have an account.".

Figure 3.21: Signup Page

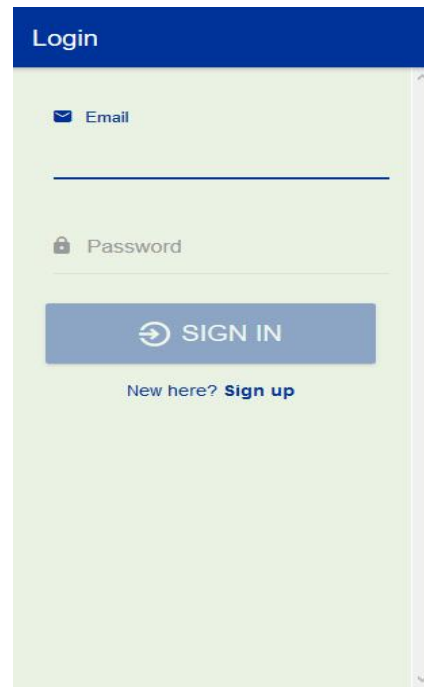
A mobile app interface for logging in. It features a blue header with the text "Login". Below the header, there are two input fields: "Email" with an envelope icon and "Password" with a lock icon. At the bottom, there is a blue "SIGN IN" button with a circular arrow icon and a link that says "New here? Sign up".

Figure 3.22: Login Page

### 3.8.3 Facial Registration and Default Location

Figure 3.23, on the left, is the face registering page. Here the user has to upload a 10 seconds video of his or her face, which will later be used by the facial recognition module to train itself.

Figure 3.24, on the right, shows the add default location page. The user has to add his or her default physical location, which indicates where the user will be found, and will be used when receiving packages.



Figure 3.23: Register Face

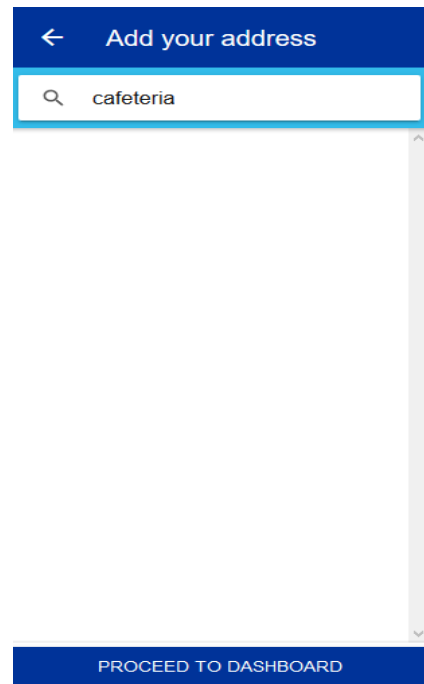


Figure 3.24: Add Default Location

#### 3.8.4 User Dashboard and Delivery History

Figure 3.25, on the left, shows the user dashboard. This is the first screen the signed in users will see once they open their app. It shows the live ongoing delivering, as well as, contains the buttons to initiate a new package delivery, or view previous deliveries.

Figure 3.26, on the right, shows the delivery history. It shows the history of all the past sent, and received packages. User can click a booking to view its detail.

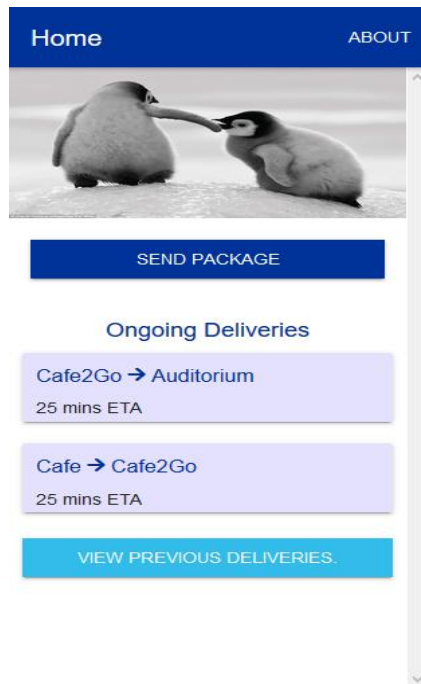


Figure 3.25: Dashboard



Figure 3.26: History

### 3.8.5 Booking Details and Pickup Point

Figure 3.27, on the left, shows the booking details page. It shows the details of the selected booking. The booking details include, sender, receiver, data, time sent, and time received.

Figure 3.28, on the right, shows the pickup point page. It is the first step when sending a package. The sender has to add their address so that the robot can come to his or her location to pick up the package.

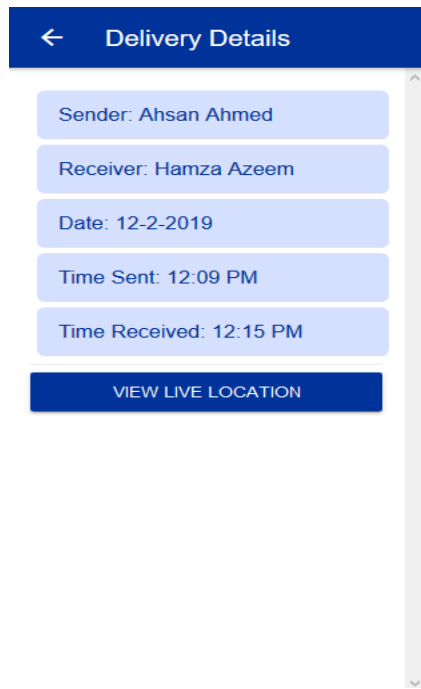


Figure 3.27: Booking Details

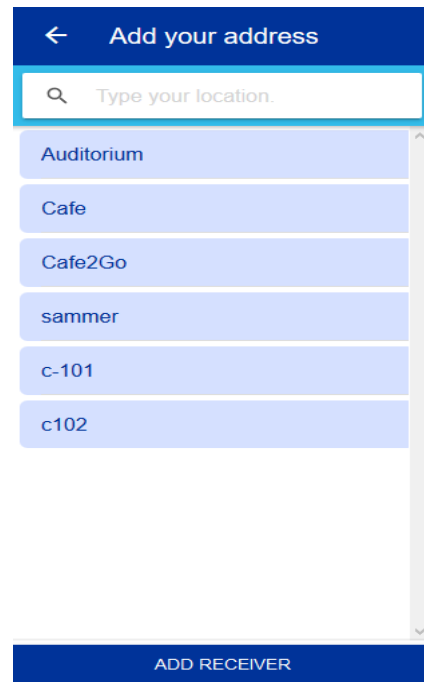


Figure 3.28: Pickup Point

### 3.8.6 Add Receiver and About Page

Figure 3.29, shows the add receiver page. This is the second step when sending a package. The user has to tag their receiver, so that the robot can go to the receiver's default location to deliver the package, and inform the receiver that his or her package has arrived.

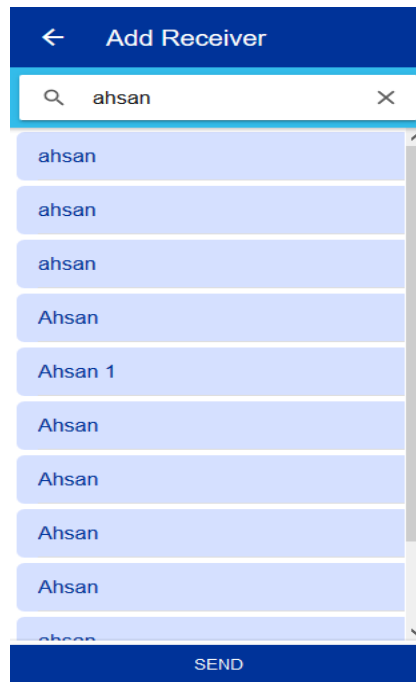


Figure 3.29: Add Receiver Page

A tutorial for the Android application can be found in Appendix D.

### 3.9 Web Application

#### 3.9.1 About

The web app is used to keep a track of the bookings that come from the app. It is also used to visualize the data in the database, for example, user information. It also allows the admin to add points on the map, and extract the data that is on the database in an excel sheet.

#### 3.9.2 Login Page

Figure 3.30 shows the admin login page. This is the page that the system admin can use to access the main system.

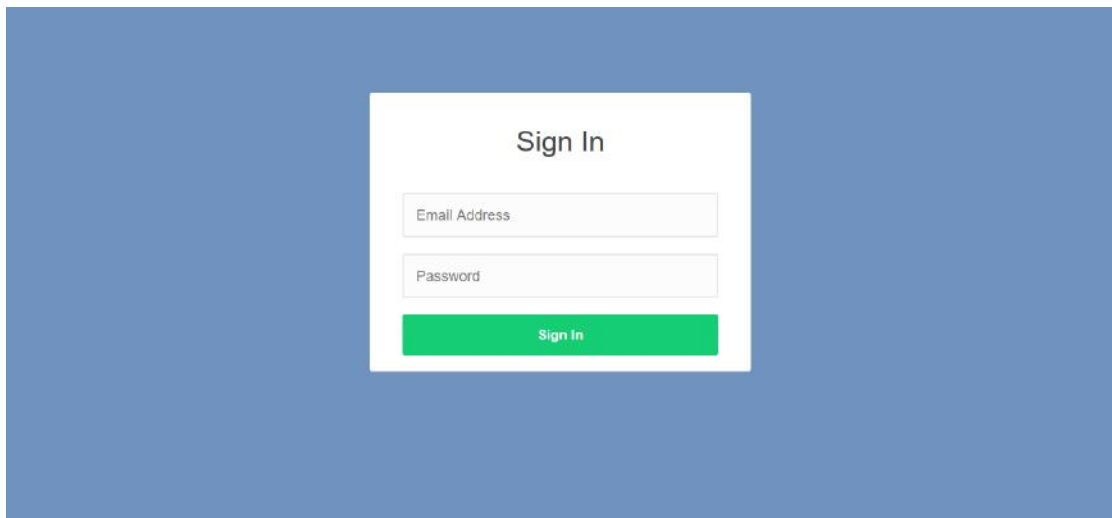


Figure 3.30: Admin Login

### 3.9.3 Dashboard Page

Figure 3.31 shows the admin dashboard page. This page allows the admin to interact with the database, and see the users of the apps, along with delivery reports, and add pointers to the generated map.

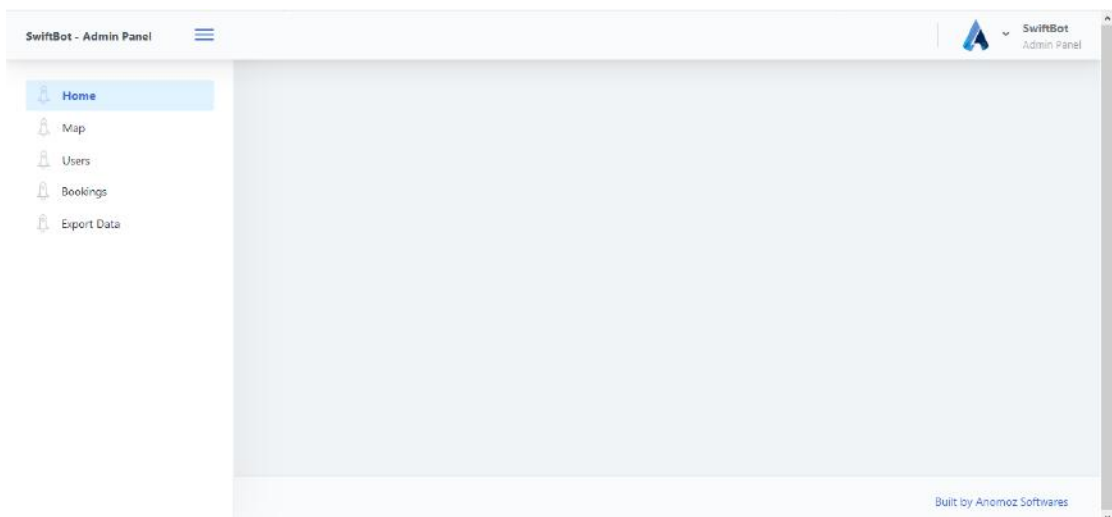
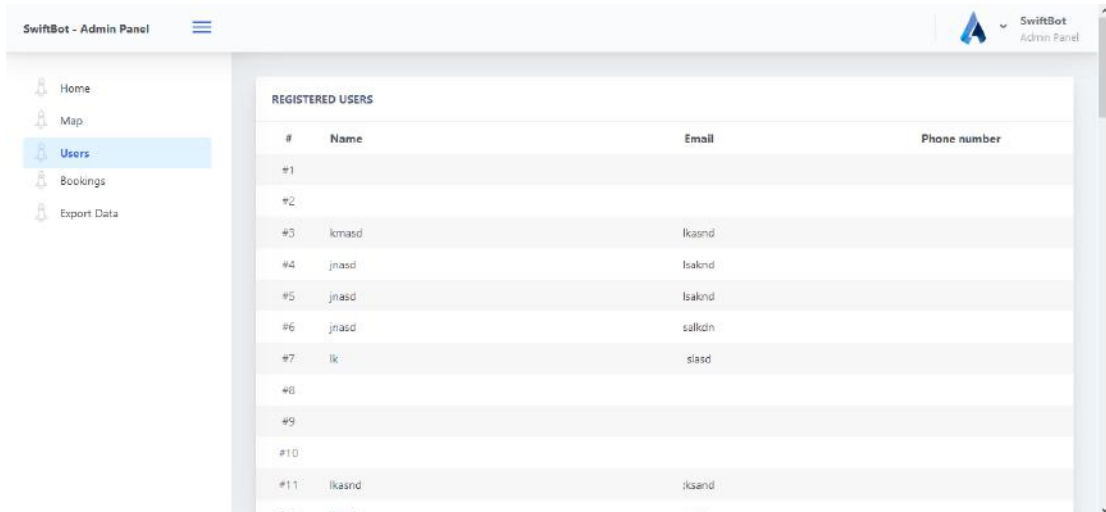


Figure 3.31: Admin Dashboard

### 3.9.4 Users Table

Figure 3.32 shows the users table. It contains the list of all the registered app users, along with their name, email, and other information.





#	Name	Email	Phone number
#1			
#2			
#3	lmasd	lksand	
#4	jnasd	lsaknd	
#5	jnasd	lsaknd	
#6	jnasd	salkdn	
#7	lk	slasd	
#8			
#9			
#10			
#11	lksand	jksand	

Figure 3.32: Users Table

### 3.9.5 Floor Map Editor

Figure 3.33 is the floor map editor, one of the most important parts of the whole system. The admin can add map points on the generated LIDAR map, so that the users can use these map pointers to send, and receive packages across the indoor space. It allows the admin to add map pointers, and update the existing pointers. It is important to know that, as the map image is of different type than the one on ROS, hard coded ROS map values were supplied into these pointers. It is not possible to transform coordinates on this map to the ones on ROS, as there seems to be no mathematical relationship between them. Therefore, every time the coordinates were put up on these maps, it was required to first find out what they are corresponding to on ROS.

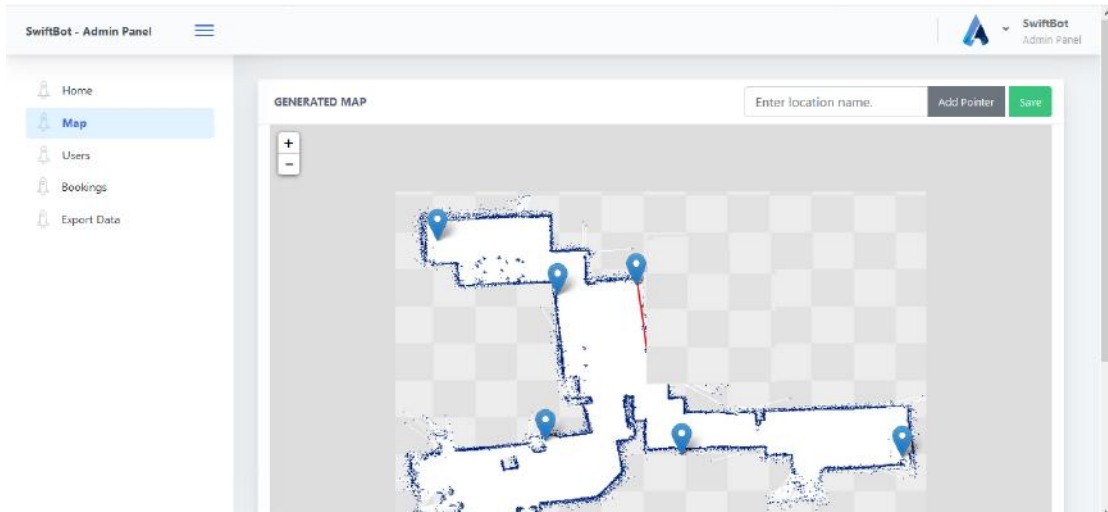


Figure 3.33: Floor Map Editor

### 3.9.6 Export Data

Figure 3.34 shows the export data page. It allows the admin to export stored data from the database to the excel files, for further data analysis.

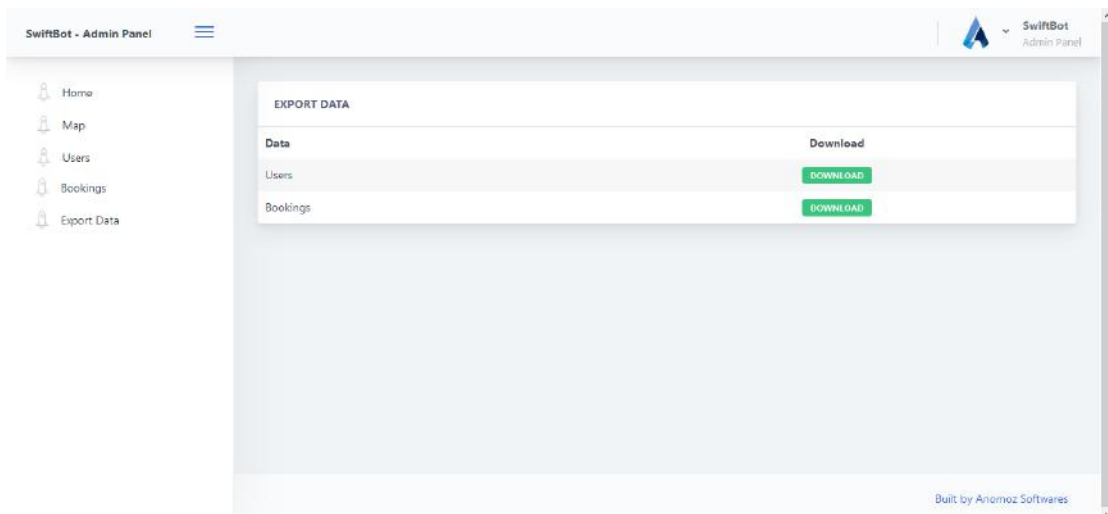


Figure 3.34: Export Data

Figure 3.35 shows the export to Excel page. This is the page from where the admin can download the generated excel file.

Registered Swift Users			
Export to excel			
No.	Name	Email	Phone Number
67	Hayyan Niamatullah	8912dx@gmail.com	1273691273
66			
65	Hshs	yshshsbdb	Hshs
64	aosdjln	lknsad	lkesnd
63			
62	Gsbs	hsbd	1e8e6b287f7d2b7d
61	ahsan	snahmed1998@gmail.com	
60	Ahsan	snahmed1998@gmail.com	
59	asd	asdasd	
58	asdlasdlk	askldn	
57			
56			
55			

Figure 3.35: Export to Excel

## 3.10 Servers

### 3.10.1 Central Server

The central server, as the name suggests, is responsible for all the communication between the different modules of the system. It provides different communication channels, including connection to the app, facial recognition server, and the robot.

### 3.10.2 Load Balancing Server

The load balancing server is present to provide assistance to the Raspberry Pi in handling the computations required by the robot to operate. Due to the Raspberry Pi's limited computational capabilities, it is unable to execute certain control loops at the desired frequency, and hence sharing some of the more computationally expensive programs of the robot to an external server becomes necessary. In this scenario, the load balancing server is nothing but a laptop, running the Ubuntu operating system with ROS installed on it as well. The more resource intensive programs, including the path planning program, is executed on this server, while the rest of the programs are executed on the Raspberry Pi. The communication between the Raspberry Pi, and the load balancing server occurs over WiFi, using an ssh connection.

## **CHAPTER 4**

### **PROTOTYPING AND TESTING**

#### **4.1 Motors**

One of the fundamental requirements of practically implementing a differential drive system is the locomotion mechanism i.e. the motors. The main requirement for the motors in this project is to be easily controllable, so that speeds and directions can be changed promptly as required by the path planning algorithm, as well as, to provide enough torque to bear with the weight of the system. With the two layers of the robot, as well as, a maximum payload of 1 kg, the mass of the robot requires powerful motors that can provide sufficient traction to operate without too much stress and current draw. For these reasons, geared DC motors appeared as the ideal choice of serving as the actuators of the system. A geared DC motor works on the basis of conservation of angular momentum by transferring the energy between differently sized gears, via appropriately selected gear ratios to provide the desired torque. The consequence of this mechanism is that the speed of the motor gets reduced as the torque and speed follow inverse relationships [36]. The motor selected for this project is a 12 V, 70:1 gear ratio geared DC motor, with an attached Hall effect quadrature encoder circuit. The unit has a maximum speed of 150 RPM at no-load, operating at 200 mA at this condition, and it possesses a maximum power consumption of 12 W. The recommended value of a continuously applied load on the motor is 10 kg·cm with a recommended upper limit for the instantaneous torque at 25 kg·cm, after which the unit may experience adverse thermal damage of windings and brushes.

To ensure that the encoder circuit is working as required, an oscilloscope was used to observe the signal output from the two encoder channels for each motor. The idea is to obtain two square wave pulse trains which are around 90° out of phase with each other. After applying a voltage of 12 V to one of the motors, and connecting oscilloscope

leads to the outputs, the graph in Figure 4.1 was obtained.



Figure 4.1: Encoder Pulse Output

The next order of testing was to monitor the difference in speed between the motors being used for each of the wheels. For a dry run, both motors were supplied with the same DC voltage source of 12 V from a bench power supply, and the motion of the motors was observed via fiducial markers attached to the motor shafts through pieces of tape. One motor observably ran faster than the other. Another basic visual test was performed by attaching both the motors to a two-wheel robot base from a hobbyist set, and observing the subsequent motion of the unit after supplying the motors with the same voltage. The moving base produced an observable drift in one direction and hence, the speeds of the motors were determined to be uneven. It is understood that mechanical differences between the internal components of the motors, such as, physical differences in the constituent motor windings can cause them to run at different speeds, even when the same voltage is applied to them so, in order to confirm this, some data was collected.

A simple timer based loop was implemented on an Arduino to measure RPM every

few milliseconds using the encoder signals from the motors (Appendix F.1). The voltage applied to both the motors was varied over a period of time and using the Serial Plotter tool of the Arduino IDE, a plot of the difference between motor RPMs over time was obtained as shown in Figure 4.2

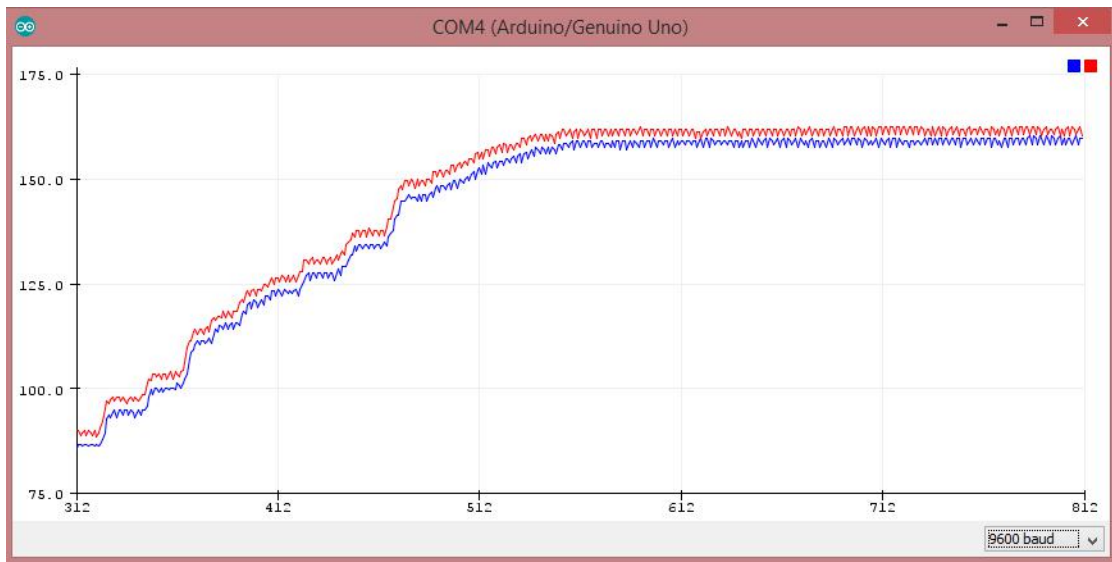


Figure 4.2: Motor RPMs Over Time

As can be observed from Figure 4.2, there is a noticeable difference in speeds between the two motors. To better quantify the difference, the RPM values being serially communicated by the Arduino to the PC were captured using a Python based utility tool called puTTY. puTTY is a tool used for monitoring serial data traffic, and it is used to collect the data being transferred over the COM port by the Arduino over a fixed period of time, and saved in a text file as indicated by the settings in Figure 4.3

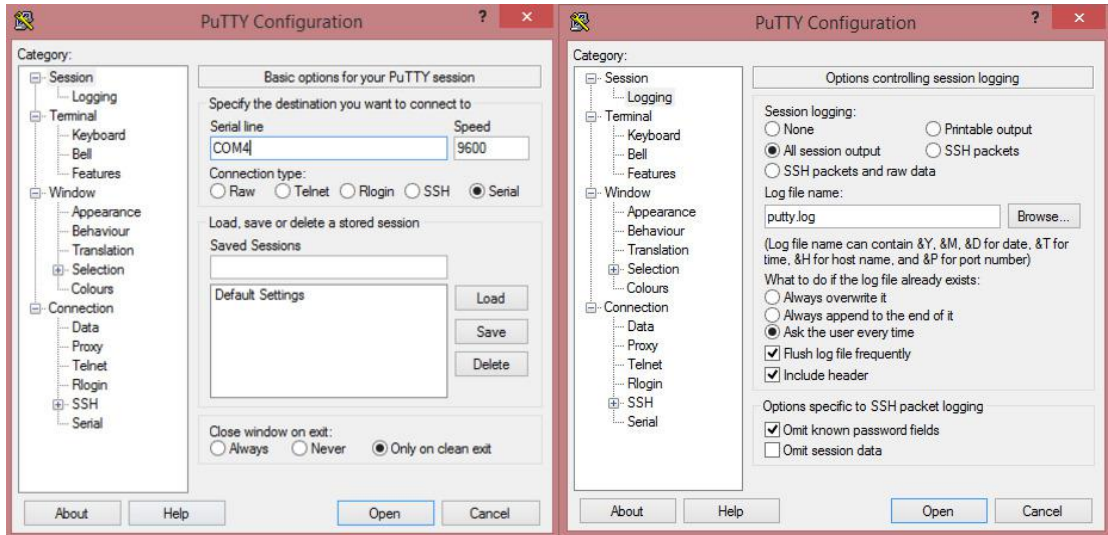


Figure 4.3: puTTY Configuration for Serial Data Collection

Since, the motors would eventually be run using the motor driver (Pololu VNH5019 Dual Motor Driver), it was decided to test the speed difference between the motors, via operation through the motor driver, as well as, by supplying power directly via a bench power supply. This would enable the observation of the actual response of the motors when they would be running as the part of the completed prototype. The two tests were then performed by supplying 12 V (to match the voltage that the battery would be supplying) from the bench supply first directly, and then through the motor driver, and the data from each run was collected. After collecting the data, a Python script was written to format the data and save it into a csv file for further analysis. The script is present in Appendix G.1. The trends in the data were plotted, and the mean values of the left and right motor speeds were computed over an interval as shown by the plots in Figures 4.4 and 4.5, and the tables 4.1 and 4.2.

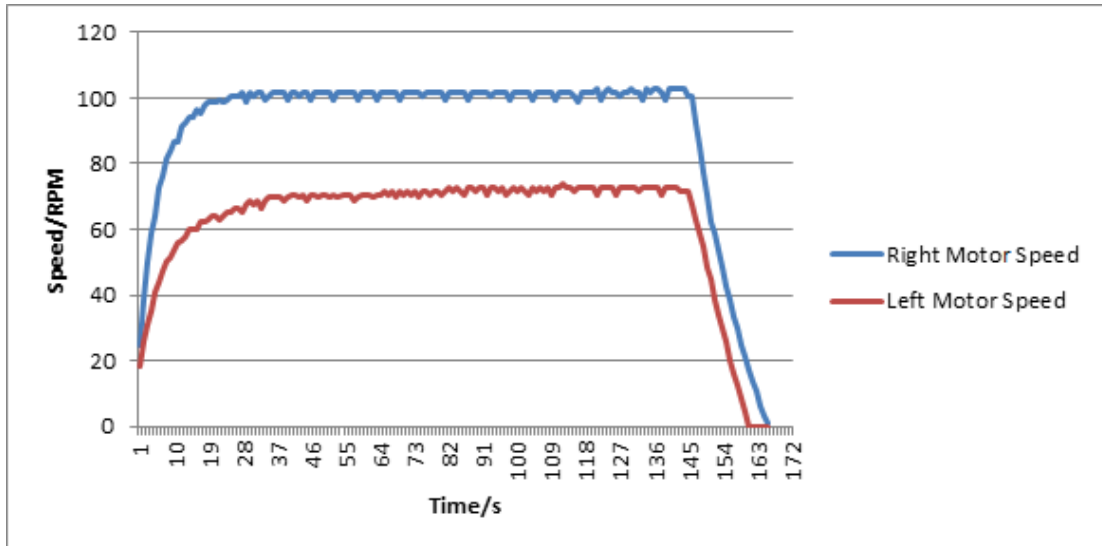


Figure 4.4: Motor Speed Comparison - Motor Driver

Table 4.1: Average Speed Difference - Motor Driver

Right Speed Average (RPM)	Left Speed Average (RPM)	Average Difference (RPM)
101.3	71.2	30.0

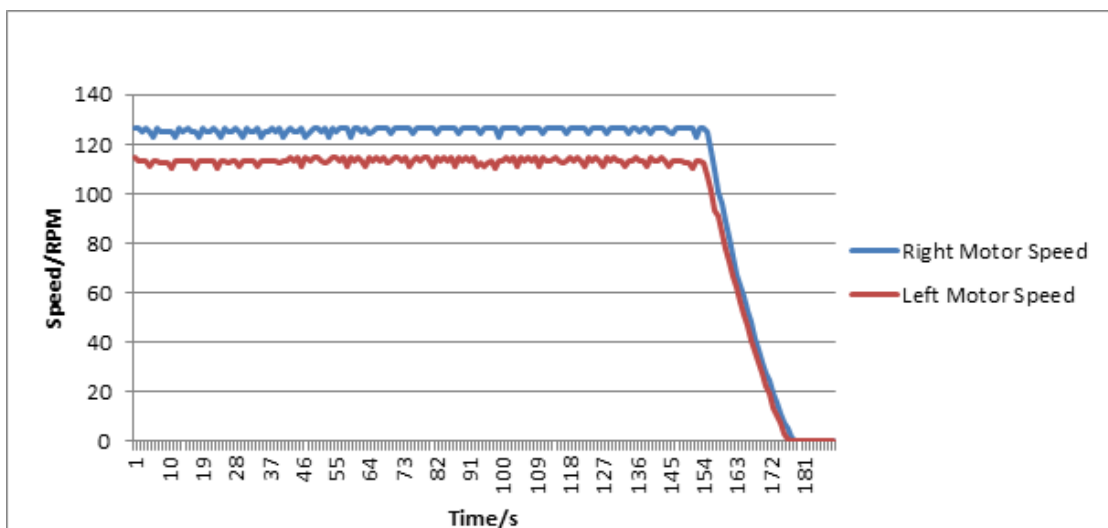


Figure 4.5: Motor Speed Comparison - DC Supply



Table 4.2: Average Speed Difference - DC Supply

Right Speed Average (RPM)	Left Speed Average (RPM)	Average Difference (RPM)
125.4	113.0	12.4

As can be seen, the average difference between the rotational speeds of the motors was around 12 RPM when not using the motor driver board, and around 30 RPM when using the shield, and this was recorded at the same voltage of 12 V.

Based on this information, the subsequent task involved configuring a PID controller to match the motor speeds appropriately, as required by the high level controller driving the robot.

## 4.2 PID Control

Having found that the motors move at different speeds at the same voltage input, due to inherent physical differences, some form of closed loop control system was required to ensure that the robot reaches its goal and moves smoothly towards its destination, without drifting off course. Using the output signals of the encoder motors to determine their speeds, it was decided to implement variations of PID controllers to observe locomotion performance of the robot. The initial testing was carried out on the first prototype, and the PID algorithms were programmed directly on the Arduino Uno that was controlling the motors. This was done in order to have the controller closer to the hardware to ensure that the system has minimal timing issues. The speed of the motors is controlled via Pulse Width Modulation (PWM), and for the Arduino Uno, the timing operation to implement PWM is carried out by using the `millis()` function primarily, which uses the 8-bit Timer 0 of the Arduino Uno. The `millis()` function returns the number of milliseconds since the function is called, and hence a timer interrupt is implemented by using this function to count the number of output pulses from the encoder motors for a fixed period of time. The output pulses themselves are read by triggering hardware interrupts built in the Arduino Uno. The number of pulses measured per unit

time are used to determine the speeds of the motors for comparison.

A particular implementation of a Proportional-Integral (PI) controller was applied as given in Appendix F.2. At this stage, the checks performed were simply on the basis of observation, by comparing motor output speeds with each other on the Serial Monitor of the Arduino IDE, and by observing the movement of the prototype on the lab floor with respect to a straight line marked on it with tape. The tuning of the PI algorithm at this point was also on the basis of the observation of the robot's movement with the adjustments of the proportional and integral constants being based on these observations. Due to the minimal effect of the derivative constant on improving the robot's motion, it was set to zero.

For the final prototype, the robot's controller is implemented using ROS through the differential\_drive package. This is done in order to more easily integrate the various subsystems, such as, obstacle detection, mapping, and navigation with the differential drive controller. The use of the differential\_drive package enables interfacing with the navigation stack (the overarching navigation controller), so that the differential drive system takes in twist messages from the navigation stack, and outputs the left and right wheel strengths which are converted into PWM signals for the motors, via the code implemented on the Arduino. Implementing the differential drive controller via ROS also gives the advantage of making the controller independent of the hardware platform being used. To exemplify this, the functionality of the PI controller in Appendix F.2 can be implemented via a ROS package. Hence, this functionality can be ported to another low level microcontroller system as required, thus reducing the dependence on hardware specific code for the controller. The code implemented on the Arduino serves as an interface to receive the wheel strengths coming from the differential\_drive package, output them as PWM signals for the motors, and transmit back odometry data in the form of number of encoder pulses received during an interrupt event. The interfacing code implemented on the Arduino can be found in Appendix F.3.

A script to configure the variables, namely the proportional ( $K_d$ ), integral ( $K_i$ ), and the derivative ( $K_d$ ) elements of the PID controller is available as part of the differen-

tial\_drive package, and thus, for the final prototype, the tuning of the PID algorithm is achieved through the variation of these parameters. One limitation observed while using the ROS based controller was the spiking of the PWM values due to the controller frequency missing its desired rate of operation. A test run of the ROS controller was performed on the first prototype by setting the Kp, Ki, and Kd values to those observed from a previous capstone project that used the same motors, and a similar ROS controller [37]. The parameters in the PID configuration file were set to the values, Kp = 480, Ki = 250, and Kd = 18, as shown in Figure 4.6.

```

<launch>

<arg name="namespace" default="/" />
<arg name="tfpre" default="$ (arg namespace) " />
<rosparam param="ticks_meter">17825</rosparam>

<node pkg="differential_drive/scripts" type="twist_to_motors.py" name="twist" output="screen">
  <remap from="twist" to="cmd_val" />
  <remap from="lwheel_vtarget" to="left_target" />
  <remap from="rwheel_vtarget" to="right_target" />
  <rosparam param="base_weight">0.195</rosparam>
</node>
<node pkg="differential_drive" type="pid_velocity.py" name="lpid">
  <remap from="wheel" to="left_encoder" />
  <remap from="motor_cmd" to="left_pwm" />
  <remap from="wheel_vtarget" to="left_target" />
  <remap from="wheel_vel" to="left_actual" />
  <rosparam param="Kp">480</rosparam>
  <rosparam param="Ki">250</rosparam>
  <rosparam param="Kd">18</rosparam>
  <rosparam param="outmin">255</rosparam>
  <rosparam param="outmax">255</rosparam>
  <rosparam param="rate">40</rosparam>
  <rosparam param="timeout_ticks">4</rosparam>
  <rosparam param="rolling_pts">5</rosparam>
</node>
<node pkg="differential_drive" type="scripts/pid_velocity.py" name="rpil">
  <remap from="wheel" to="robot1_right_encoder" />
  <remap from="motorcmd" to="robot1_right_pwm" />
  <remap from="wheel_vtarget" to="robot1_right_target" />
  <remap from="wheel_vel" to="robot1_right_actual" />
  <rosparam param="Kp">480</rosparam>
  <rosparam param="Ki">250</rosparam>
  <rosparam param="Kd">18</rosparam>
  <rosparam param="out_min">255</rosparam>
  <rosparam param="out_max">255</rosparam>
  <rosparam param="rate">40</rosparam>
  <rosparam param="timeout_ticks">4</rosparam>
  <rosparam param="rolling_pts">5</rosparam>
</node>
<node pkg="differential_drive" type="scripts/diff_tf.py" name="edemote" output="screen">

```

Figure 4.6: PID Parameters

To observe the effects of the tuning, the motion planner of the robot was run, and a target goal was given to the robot on a loaded map of a small area in a house, given in Figure 4.7.



Figure 4.7: Map of Test Area for Prototype 1

The target speed data sent by the motion planner using the PID parameters, as well as, the data of the actual speeds recorded using the encoders were collected serially using puTTY, and the resultant data points were used to plot graphs of how the actual speeds of each of the motors responded to the target inputs, as shown in Figures 4.8 and 4.9.

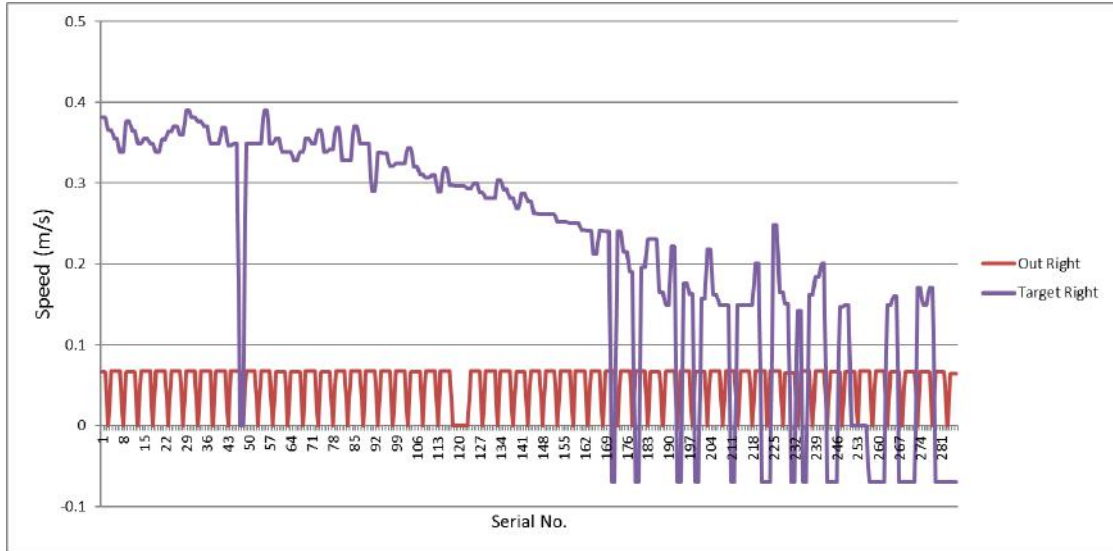


Figure 4.8: Right Wheel Target and Actual Speeds

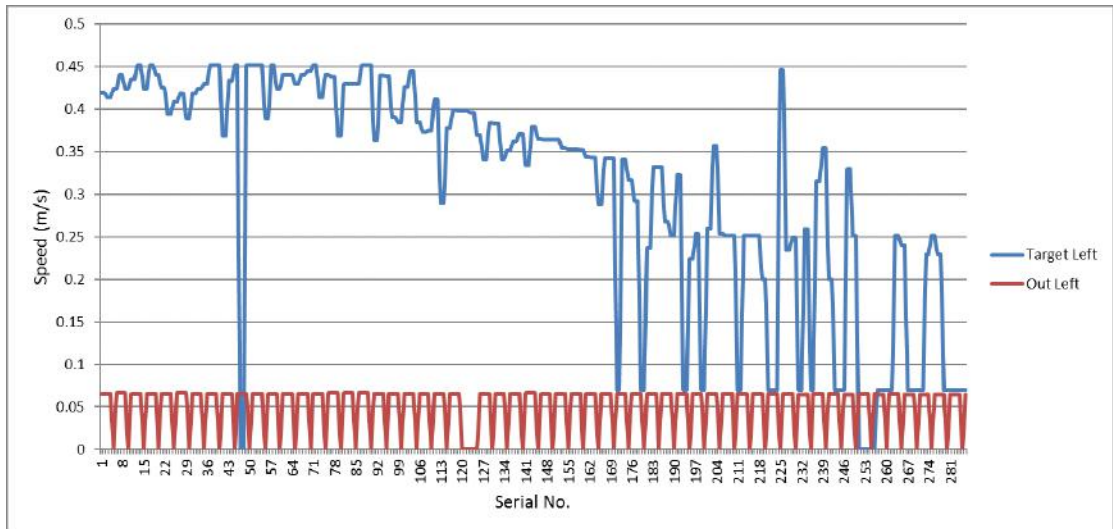


Figure 4.9: Left Wheel Target and Actual Speeds

As can be observed from Figures 4.8 and 4.9, the target velocity follows an erratic pattern. This is because the motion planning algorithm is periodically updating its values to drive the robot towards its goal. The expectation was for the motion planner to set a smooth, and more or less a level output to drive the robot in a straight line. However, the values assume the forms shown in the graphs due to how the actual output speeds behave. Observing the output speeds, they follow a periodic pattern of almost constant speeds followed by dips to zero. These consistent zero values occur due to the ROS controller missing its desired frequency rate of update. Additionally, the levels

of both the speeds are different, with the right speed moving at a marginally higher speed than the left one, which results in the robot drifting off its course. Subsequently, the motion planner sets new target speeds to guide the robot back on course towards its goal. The actual speeds also do not come anywhere near the the targeted speeds. This result implies the requirement of further tuning of the PID parameters, as well as, a tweaking of parameters, particularly those associated with navigation, in the ROS configuration files to make sure that the erratic behavior is reduced.

Being cognizant of the limitation in computation power of the Raspberry Pi, the spiking pattern was anticipated, and with this in mind, the PID algorithm was tuned. A constant speed velocity command was set to each of the wheels from the move\_base path planner, via the cmd\_vel topic. Some inspiration for the PID tuning process was taken from the Zeigler-Nicholls tuning method, whereby all the constants were set to zero, and the proportional component was slowly increased, and the level of oscillation was observed. However, the accuracy of the results could not be ascertained in a satisfactory manner due to this process (Zeigler-Nicholls), being more suited to analog controllers. Also, the ROS controller's frequency kept varying with each update cycle, and hence, the consistency of the updates could not be maintained. Table 4.3 highlights the variation in the PID parameters with respect to the responses obtained in Figures, which demonstrate the results of some PID tuning experiments. Note that the vertical axis represents speed in metres per second (m/s), and the horizontal axis represents the time scale in seconds (s).

Table 4.3: PID Parameter Tuning Values

Figure	Left			Right		
	Kp	Kd	Ki	Kp	Kd	Ki
4.10	10	0	0	100	0	0
4.11	10	0	0	200	0	0
4.12	10	10	0	200	10	0
4.13	50	20	0	200	20	0
4.14	50	20	10	200	20	10

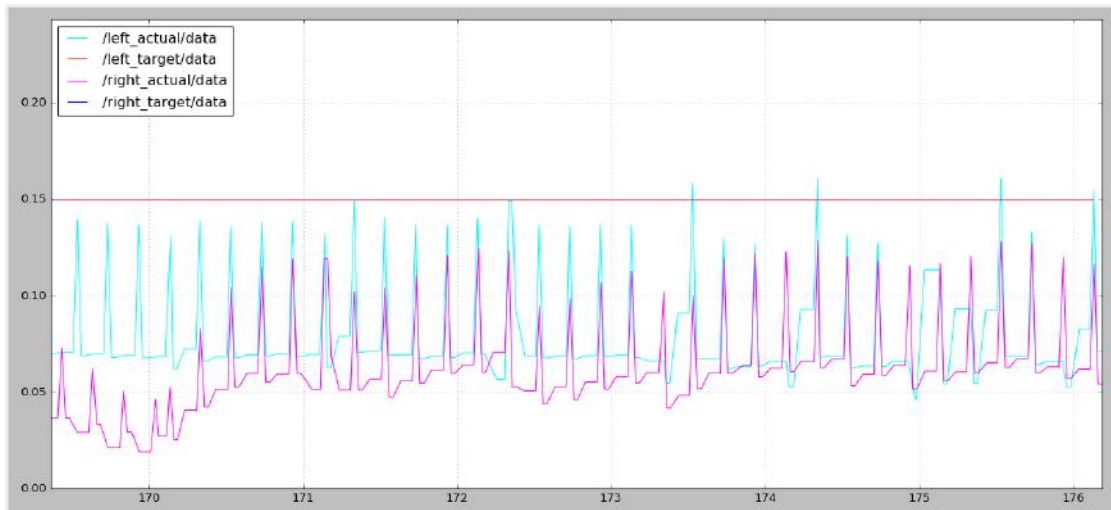


Figure 4.10: PID Tuning Result 1

With the response indicated in Figure 4.10, the speeds of both wheels do not match with each other, and the average value does not reach the setpoint, except via seemingly periodic overshoots.

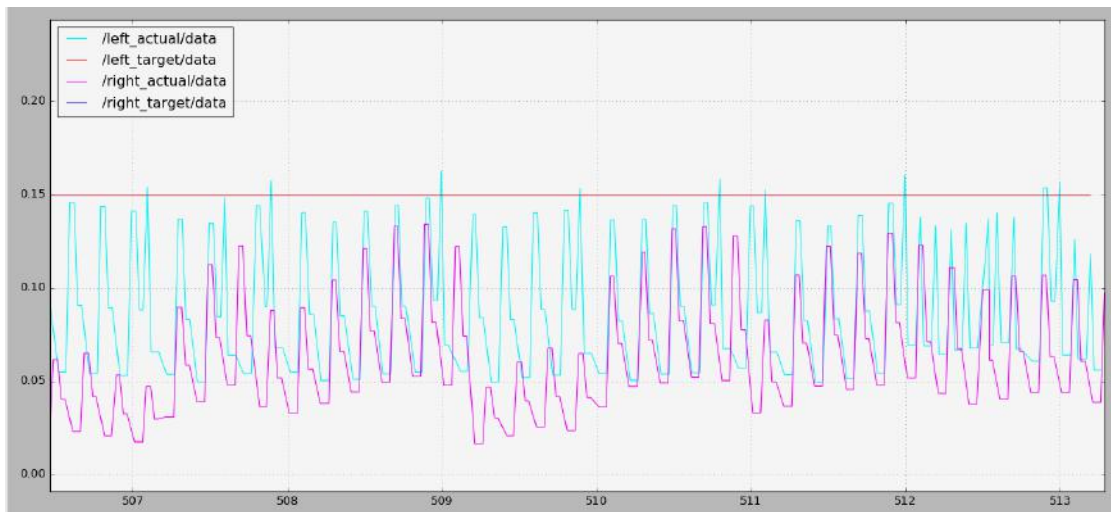


Figure 4.11: PID Tuning Result 2

In Figure 4.11, the right wheel approaches the left wheel more closely due to increase in the  $K_p$  parameter of the right wheel. However, the response is still quite uneven.

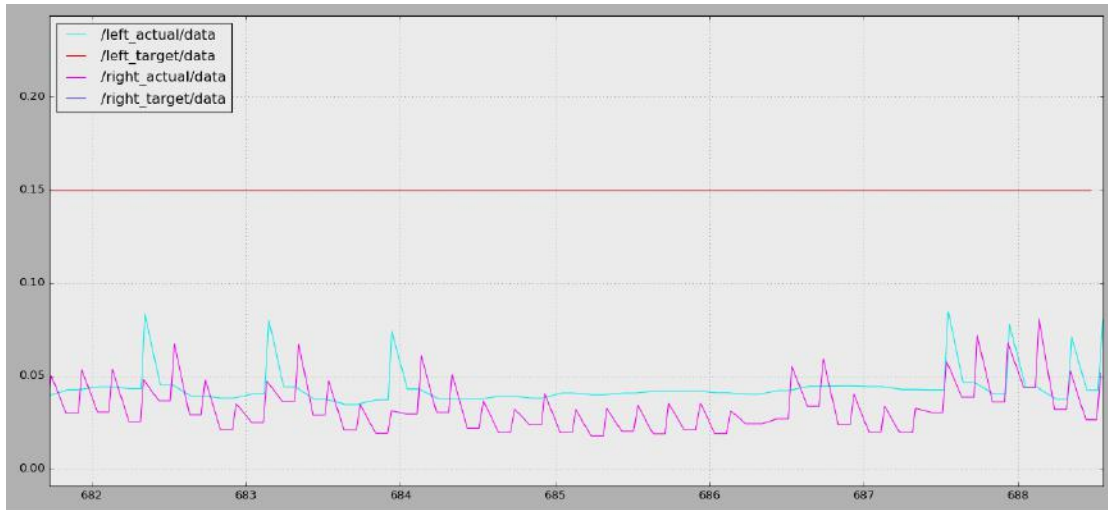


Figure 4.12: PID Tuning Result 3

By increasing the  $K_d$  parameter, for both the left and right wheels, as shown in Figure 4.12, the response becomes a bit smoother, with less spiking of speeds. However, the average values of the actual speeds still fall well below the desired setpoint.

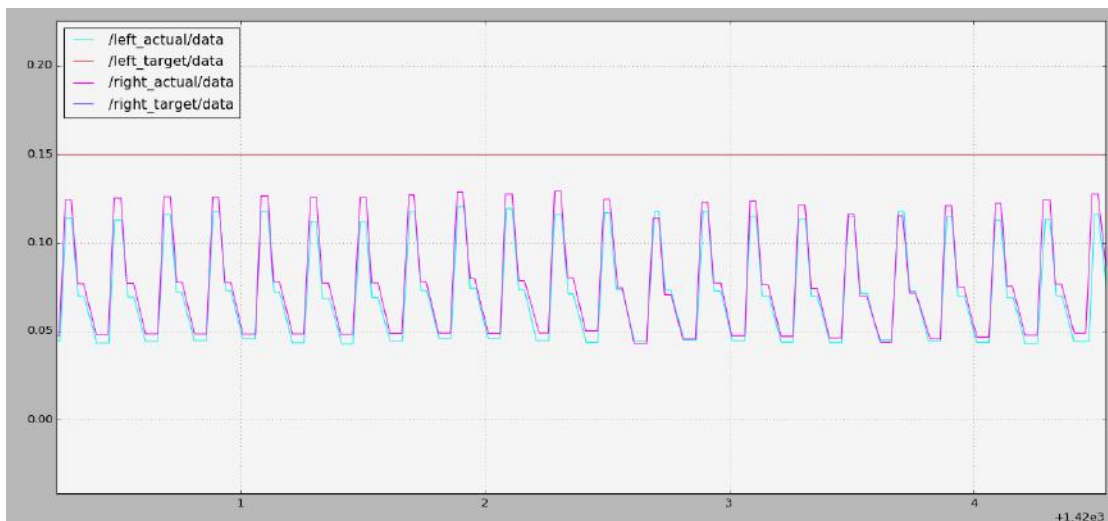


Figure 4.13: PID Tuning Result 4

The  $K_p$  parameter for the left wheel was increased, so that both the speeds match even better, as shown in Figure 4.13. The  $K_d$  parameter for both wheels was also increased to reduce any lingering overshoots. Figure 4.13 shows a marginally better result than the results achieved with the previously set parameters. Any dips of the speed to zero are also eliminated.



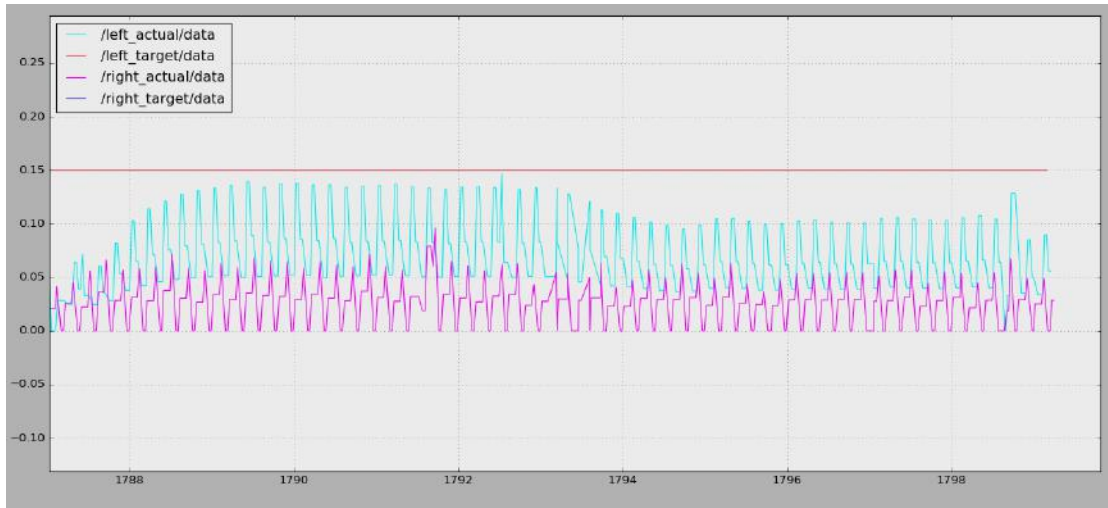


Figure 4.14: PID Tuning Result 5

While the  $K_i$  parameter was varied in many test runs, it was discovered that varying it too much often resulted in undesirable movement results, as shown in Figure 4.14, in which the speed of the right wheel drops considerably against that of the left one, and consistent zero values of the speed are also observed.

Even with various PID tuning experiments, the robot's motion did not achieve the result, as well as, that desired. For one, the amount of impulse like variations (spiking) in the speeds had to be minimized to yield a smoother motion. Additionally, it was discovered that with the existing setup, the robot's maximum speed did not exceed 0.2 - 0.25 m/s, without yielding unwanted movement. These issues were discovered to be primarily due to the low computational resources of the Raspberry Pi 3, which was not able to handle some of the more resource intensive algorithms, like the path planner in a time-effective manner. This was why the ROS controller, which was set as 10.0 Hz, kept missing its desired frequency rate, and showed values in the range of 1.0 - 2.0 Hz. The workaround to this problem was distributing some of the computational load to an external server, connected to the Raspberry Pi. The path planning program, `move_base` was hence ported to a server running the Ubuntu OS, which is connected to the Raspberry Pi, via a secure shell (ssh) connection over WiFi. Additionally, an API was programmed to communicate the values required by the necessary nodes (such as, velocity commands transmitted from `move_base` to the `differential_drive` package,

installed in the Raspberry Pi) between the server, and the Raspberry Pi. This distribution of computation served to increase the ROS controller frequency, which began to achieve rates of 7.0 - 9.0 Hz. While graphs similar to those shown in the previous Figures were still achieved, the robot's actual movement became considerably smoother. This is exemplified by Figure 4.15.

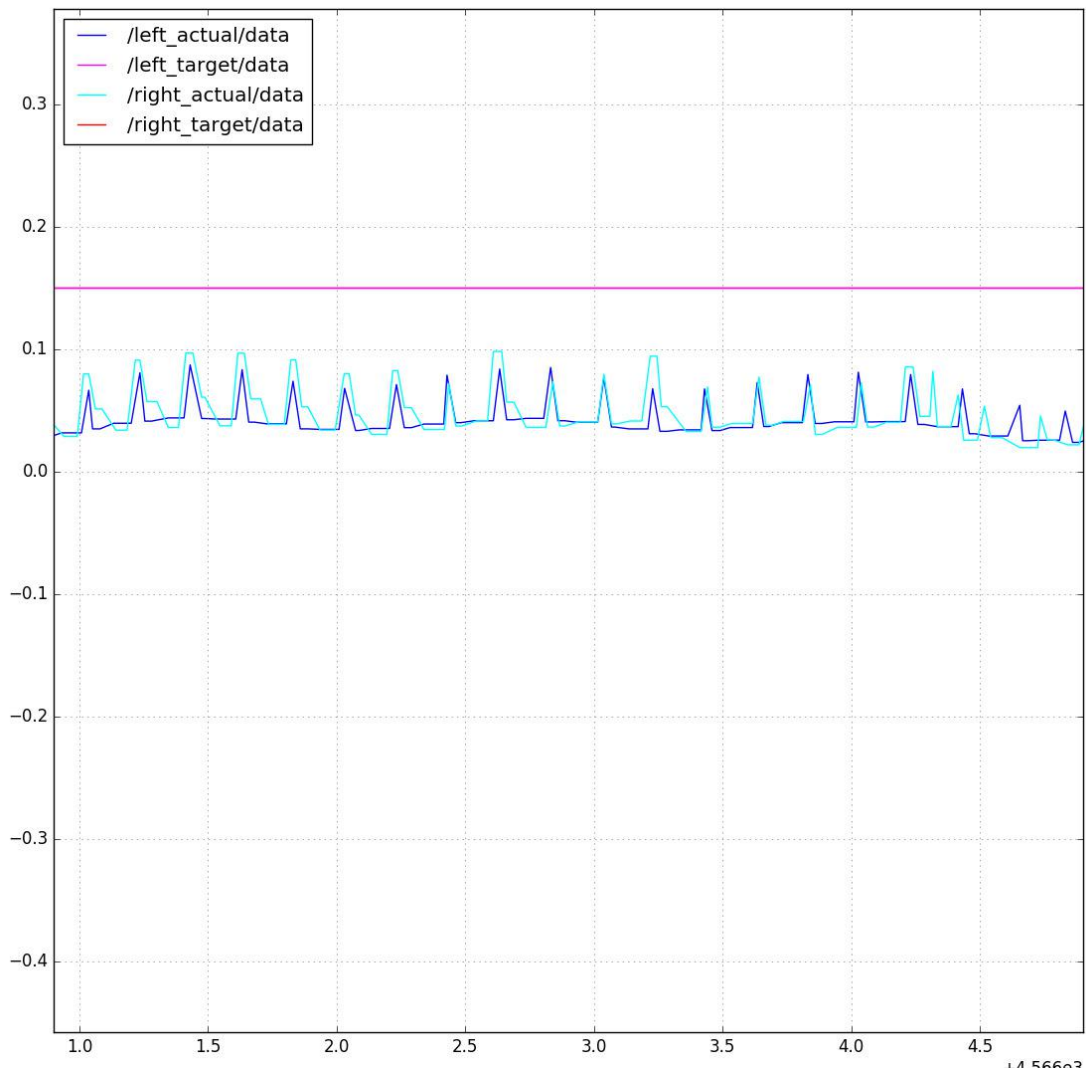


Figure 4.15: Final PID Tuning Result

### 4.3 Mapping

A variety of maps were developed to serve as testing environments for the robot. As mentioned in Chapter 3, the map development process uses Google Cartographer, an open source tool developed by Google for 2D and 3D localization and mapping. The

SLAM algorithm used by the Cartographer uses the laser range data obtained from the LIDAR to simultaneously situate where the LIDAR is, as well as, develop a map of the surroundings. Without going into too much detail of the mapping algorithm, the scope of which lies beyond this project, laser scans from the LIDAR are successively used to build up a 2D map of the surroundings. Successive laser scans are used to build up submaps, representing small chunks of the surrounding, and these are inserted into a best estimated position. A scoring system among submaps is used to construct the best estimate of the global map from localized chunks [38]. For this project, only the 2D mapping capability of Cartographer is utilized. A map of the Projects Lab at Habib University, developed using the RP LIDAR A2, and the Cartographer, is shown in Figure 4.16.

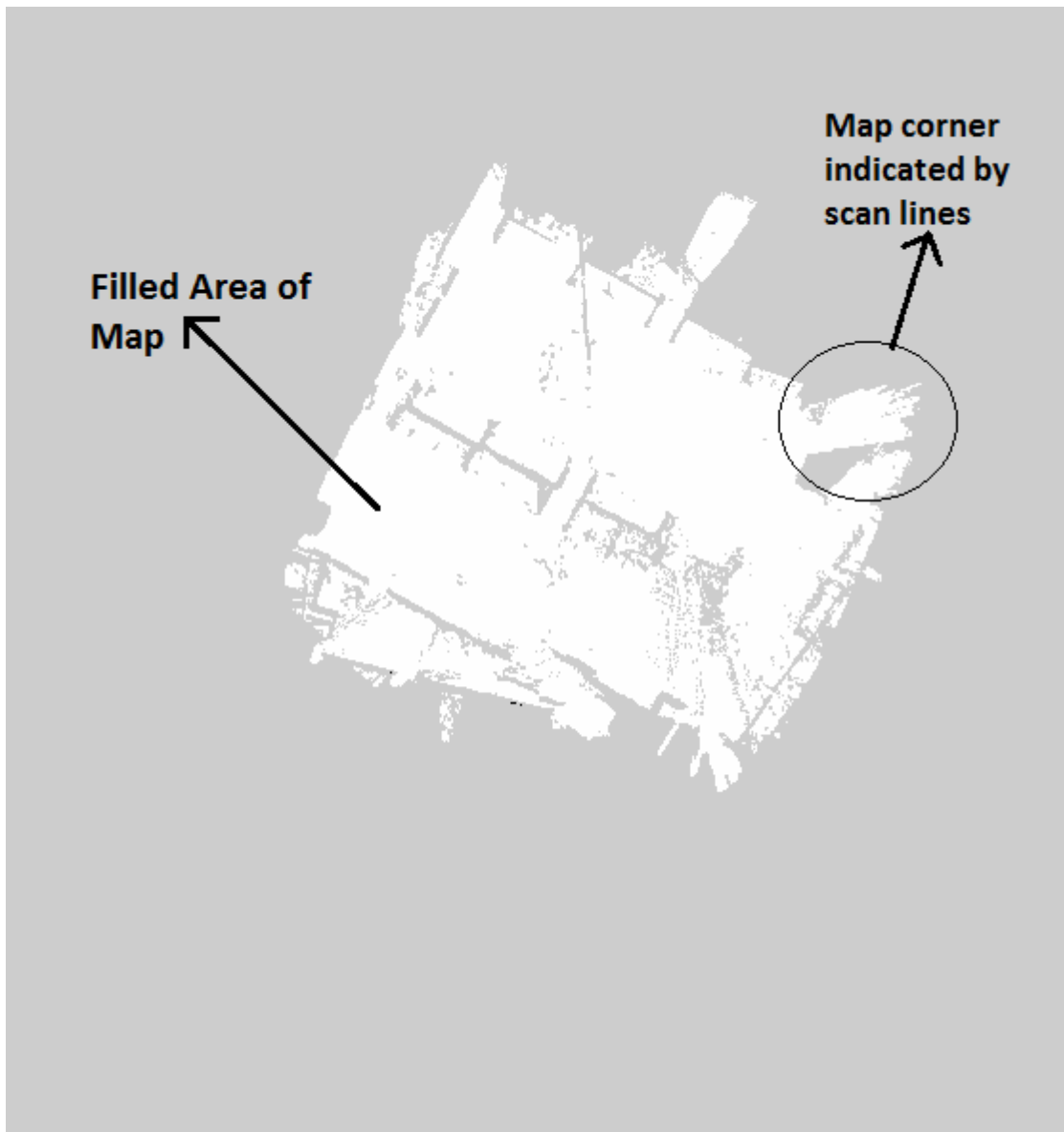


Figure 4.16: Map of Projects Lab

The area filled in, represents the scan lines where no obstructions were detected while mapping. The robot can move around freely within this area. The striations encountered at the edges of the map represent areas where the LIDAR scans could not detect any obstacles or boundaries, and hence returned scan lines. The mapping process was achieved by using the teleop node of ROS to manually move the robot around the room, and obtain sufficient scans for the LIDAR to build up a best estimate of the surrounding. An ineffective map may be of the robot when it is moved too fast through the environment, which impacts the level of detail which can be recovered from the scans, by affecting the amount of pulse data received by the LIDAR for a given area.

Rotating the robot about its position also helps the LIDAR to accumulate more detail for the area, otherwise ineffective maps can be recovered, such as, that shown in Figure 4.17, where sub-areas appear to be merging within the map.

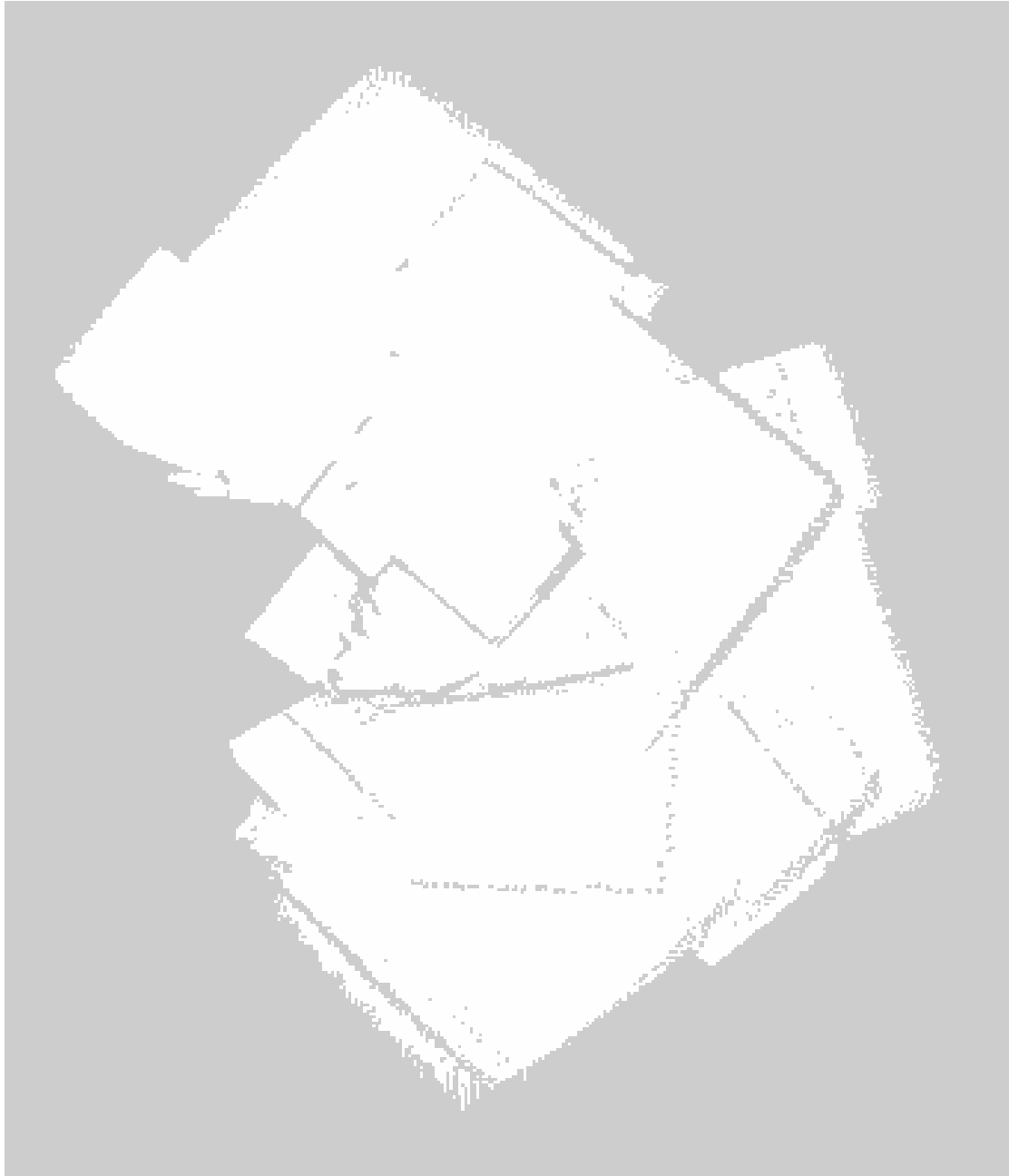


Figure 4.17: Ineffective Map

The other maps generated throughout the project are present in Appendix I. For the final testing phase as well as the demonstration, the following map was obtained and utilized, as shown in Figure 4.18:

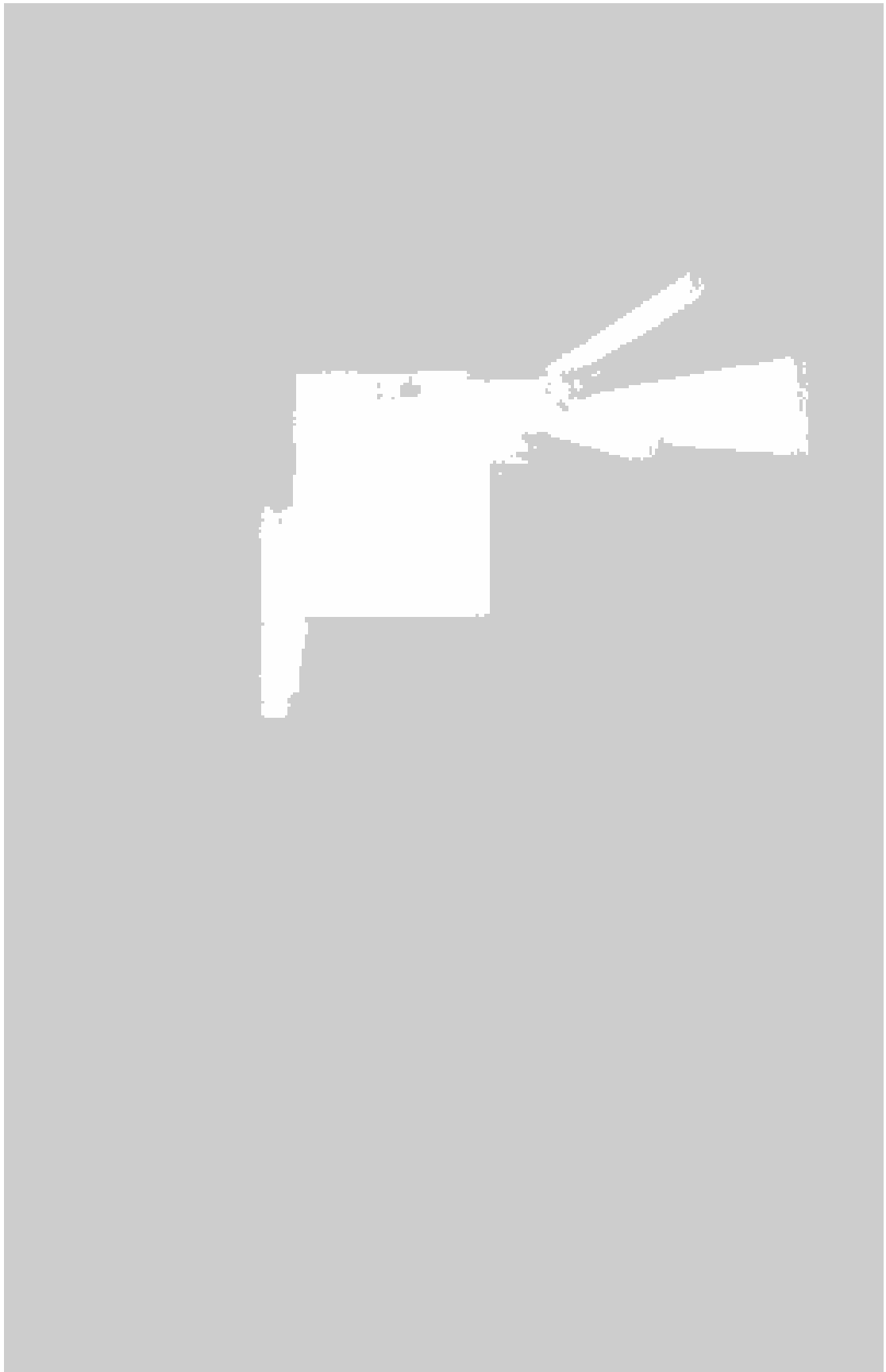


Figure 4.18: Final Map

The map shown in Figure 4.18 is of a room in a house with the scan lines extending outwards indicating an open door when the mapping process was being implemented. The start and end points for the final demonstration as appearing on the map are shown in Figure 4.19. The start point corresponds to the left side of the room upon entering, while the ending point corresponds to the entrance in front of the door.



Figure 4.19: Final Demonstration Start and End Points

#### 4.4 Laser Scanner Simulation

In order to visualize the capabilities of the laser scanner used by the robot, a simulation was developed using Gazebo, a simulation package that interfaces with ROS. The purpose was also to better visualize the obstacle detection capabilities of a laser scanner. The simulation was developed from the ground up by first defining the parameters of the robot in the Unified Robot Description Format (URDF). URDF is an XML file format used by ROS to model a robot by defining all its elements e.g. wheel radii, joints, links, general physical parameters etc. A laser scanner link was also defined, and attached with the robot. This allowed the configuration of  $360^\circ$  laser scans, allowing the observation of the laser's response with different obstacles placed around it. The purpose of this entire exercise was purely for visualization and understanding.

The following figures show some of the results of the simulations:

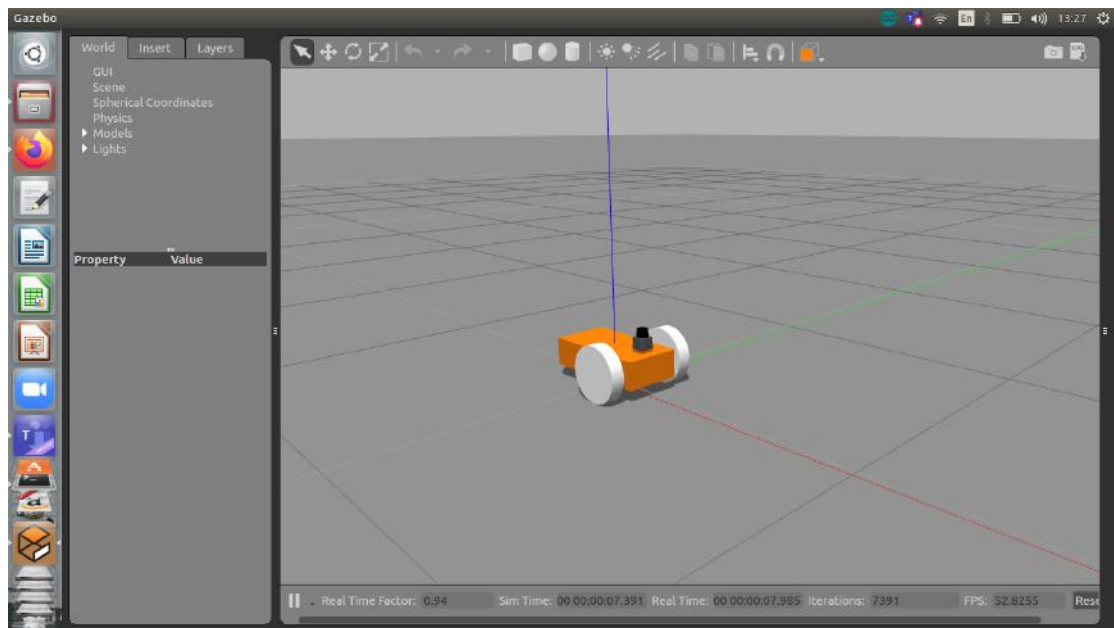


Figure 4.20: Robot Model in Simulator

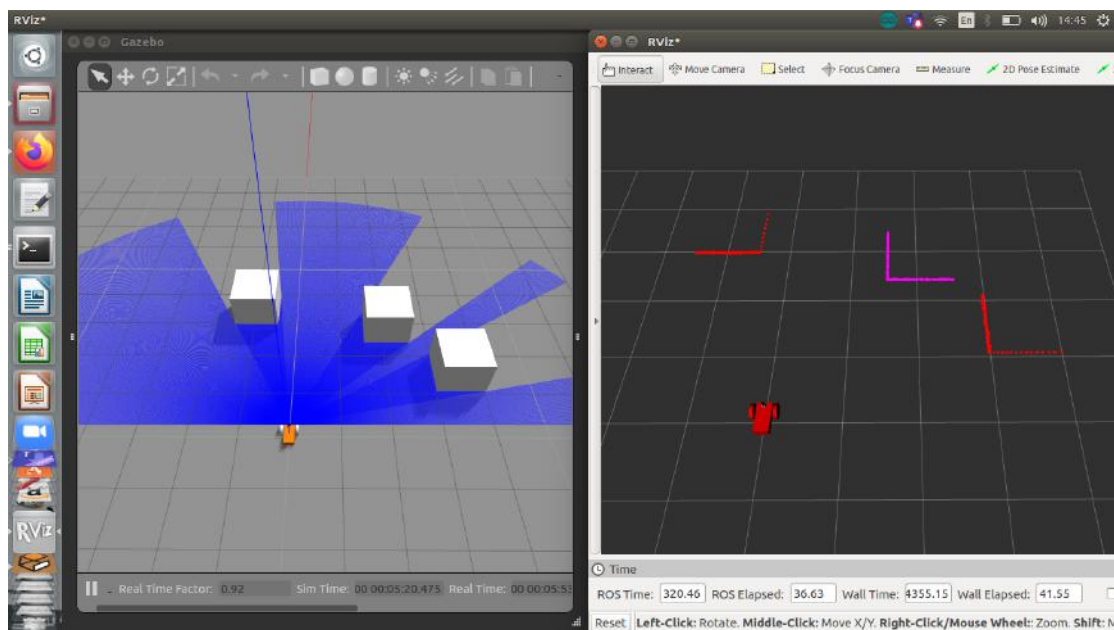


Figure 4.21: 180° Scan with Obstacles - Gazebo and Rviz Visualizations



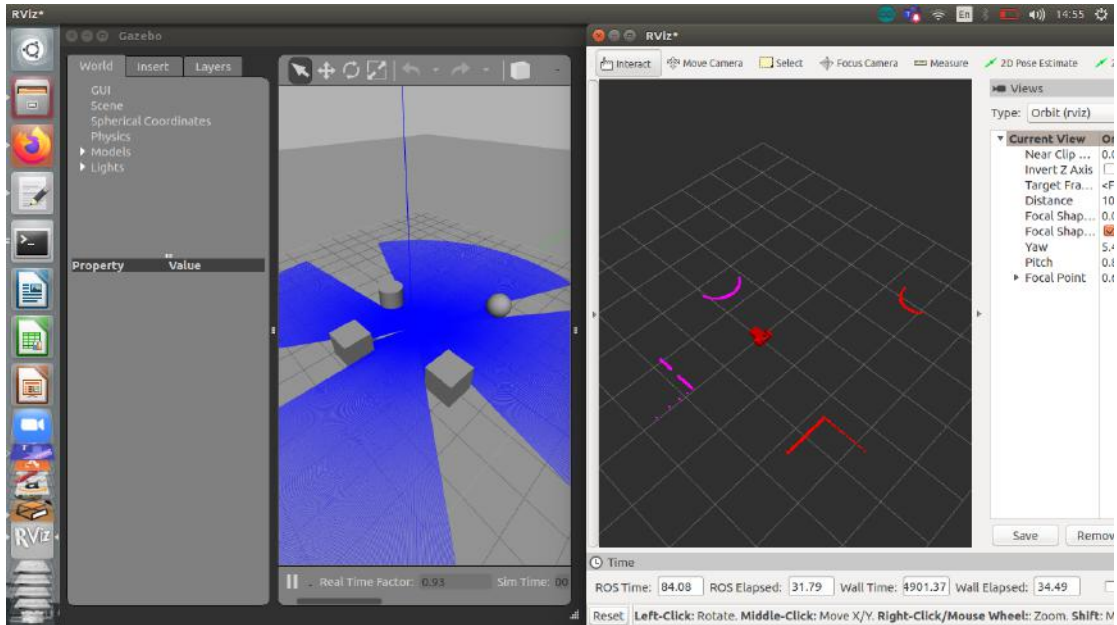


Figure 4.22: 360° Scan with Obstacles - Gazebo and Rviz Visualizations

Figures 4.21 and 4.22 show the obstacle detection simulation environment together with the concurrent visualization of the data received by the scanner in Rviz. Rviz allows the observation of the point cloud data received by the laser scanner in order to make out the edges of the obstacles detected.

## **CHAPTER 5**

### **CONCLUSION AND FUTURE WORK**

#### **5.1 Conclusion**

The fundamental goal of this project, involving the development of an end-to-end automated delivery solution, with an app based front-end, a server based back-end, and an autonomous robot, as the main delivery agent, is realized.

The key features of the robot achieved include:

1. A differential drive based robot that possesses the ability to generate a map of its surroundings, based on manual operation, and also the ability to navigate autonomously within an environment, given a known map and reference coordinates.
2. The implementation of an embedded system platform incorporating high (Raspberry Pi), and low level (Arduino Mega) controllers to actuate a robot, and receive feedback from the environment. The embedded platform is divided in the given hierarchy to separate the implementation of the higher level computationally intensive algorithms, and the interfacing of components, and data communication at the sensor level.
3. The use of a cutting edge sensor, such as, a LIDAR for environmental perception including map building, localization, and obstacle detection and avoidance.
4. The use of path planning and navigation algorithms, based on the A-star algorithm to implement autonomous navigation behavior within the robot.
5. The use of mobile and web-app development to implement a front-end to enhance the user experience, and implement a point of contact and information management for the delivery solution.

6. The use of server side technology for data management, and the implementation of seamless communication between the front-end, and the robot. The server setup also successfully facilitates the facial recognition based security feature.

## **5.2 Future Work**

A Raspberry Pi V 2.1 Camera module is used in the final prototype. This camera offers a resolution of 8 Megapixels which is not good enough for the facial recognition to function adequately. The group was not able to identify the cause of errors in the module due to the lightning conditions, either it was due to the camera, or due to the algorithm used. Also, the group did not perform rigorous testing on the facial recognition module of the project, and, as a result, there is no metric available to justify how accurate, or how reliable the module is. Looking at the future, a better camera can be installed on the prototype, particularly that of a higher resolution, and rigorous testing can be performed on the facial recognition module to make the module highly accurate, reliable, and robust according to the environmental conditions.

Going forward, the group has also realized the need to rely on multiple sensors, and sensor fusion, rather than just relying on the LIDAR for mapping, obstacle detection and avoidance etc. This would enable a more robust environmental perception system. Computer vision, a field of artificial intelligence which trains the computers to understand and interpret the visual world, is an avenue worth exploring for understanding the robot's environment. It makes use of deep learning models, videos, and cameras to help machines identify and classify objects, and consequently, react to what they see. Using cameras as additional input sensors, and employing novel computer vision algorithms can greatly reduce the chances of errors, and bring about a great deal of robustness and accuracy in the sensing functionalities of the robot. Using sensor fusion to combine the inputs of various sensors can also lead to the development of a more robust, and less error prone environmental perception model, by reducing the dependence on any one particular sensor for perception.

The Raspberry Pi model 3B used has a Quad Core 64 bit CPU with 1 GB of RAM

to handle the computational tasks, including mapping and path planning algorithms, performing high and low level communication, and handling peripherals. This amount of RAM is not enough to efficiently run some programs. As an alternative, future groups can look toward using a more powerful single board computer, such as, a Raspberry Pi 4, to handle computationally intensive tasks.

While working on the project, the team realized the importance of using simulations to test and predict hypotheses, and system behaviors, and so, going forward, the use of simulation tools can serve as an indispensable measure for testing, and relating theoretical data with real world insights. A focus on data gathering, and analysis via simulations should thus be a component to be considered for any future work.

# **Appendices**

## APPENDIX A

### BILL OF MATERIALS

Serial Number	Material/Equipment	Acquired From	Quantity Acquired	Quantity Returned	Cost/Per Piece-PKR	Total Cost-PKR
1	Pololu 12 V DC 70:1 Gear Ratio Encoder Motor	Lab	4	2	3504	7008
2	Acrylic Sheet + Laser Cutting of Bottom and Top Frames	Outside Lab	1	0	1300	1300
3	VGA Cable	Outside Lab	1	0	420	420
4	Set Screw Hub, 6 mm Bore	Outside Lab	2	0	610	1220
5	<b>L298 Motor Driver</b>	<b>Lab</b>	<b>2</b>	<b>2</b>	<b>230</b>	<b>0</b>
6	Raspberry Pi 3	Lab	2	1	6375	6375
7	5 Inch Raspberry Pi LCD	Lab	1	1	5000	0
8	Arduino Uno	Lab	1	1	840	0
9	<b>Breadboard</b>	<b>Lab</b>	<b>1</b>	<b>1</b>	<b>130</b>	<b>0</b>
10	SD Card 32 GB	Lab	2	1	710	710
11	Pololu VNH5019 Dual Motor Driver	Lab	2	1	1200	1200
12	LIPO Battery 11.1 V 3-Cell 3800 mAh	Lab	1	1	2400	0
13	Roller Ball Caster	Lab	3	3	195	0
14	USB Cable	Lab	1	0	250	250
15	Arduino Mega	Lab	1	0	1150	1150
16	<b>Raspberry Pi 5 V 3-Amp</b>	<b>Lab</b>	<b>1</b>	<b>0</b>	<b>450</b>	<b>0</b>
17	Acrylic Sheet	Lab	1	0	300	300
18	LIDAR	Lab	1	0	University Asset	NA
19	Raspberry Pi Camera	Lab	1	0	4200	4200
20	Veroboard	Lab	1	0	130	130
21	Jumper Wire + Thick Stranded Wire	Lab	-	-	Cost Not Charged	NA
22	Solenoid Lock	Outside Lab	1	0	500	500
23	<b>5 V Relay</b>	<b>Lab</b>	<b>1</b>	<b>0</b>	<b>30</b>	<b>30</b>
24	Large Brass Spacer	Outside Lab	12	0	25	300
25	Small Brass Spacer	Outside Lab	35	0	10	350
26	Heat Shrink Tubing (1m)	Outside Lab	2	0	20	40
27	Lithium Ion Battery	Outside Lab	3	0	80	240
28	12 mm M3 Machine Screw	Outside Lab	8	0	8	64
29	30 cm Raspberry Pi FFC Cable	Outside Lab	1	0	180	180
30	Lithium Ion Single Cell Charger	Outside Lab	1	0	200	200
31	6 pin JST Connector	Outside Lab	2	0	30	60
32	Personal Expense -> Locker Key Copy	Outside Lab	4	0	22.5	90
33	Ball Caster	Outside Lab	2	0	150	300
34	2 Pin Screw Terminal Block Connector	Outside Lab	11	0	10	110
35	40 Pin Single Row Male Header Strip	Outside Lab	1	0	10	10
36	L Shaped Connector	Outside Lab	2	0	800	1600
37	5.1 V 3.5 A Power Bank	Outside Lab	1	0	Personal Property	NA
					<b>Total Cost *</b>	<b>28337</b>

KEY	
	Quantity of items borrowed in extra, these items are to be returned to the lab. Their respective costs are not a part of the expense calculation.
<b>Note =</b>	For the Lab item quantities that are used in the prototype, it is expected that once the prototype is submitted to the university, their costs will be compensated.

Figure A.1: SWIFTBOT - Bill of Materials

## APPENDIX B

### ROS (ROBOT OPERATING SYSTEM) TUTORIAL

#### Files Related to ROS

(Note: All of the installations should be done on the Raspberry Pi, unless otherwise mentioned).

1. If you want to run these files, then need to first download (ROS Kinetic) from <http://wiki.ros.org/kinetic/Installation/Ubuntu> (You need to have Ubuntu installed in order to install ROS). You also need an Arduino, and the robot to make it work.
2. Once you have done that, please copy the files and folders under the (robot directory) (on github) <https://github.com/AhsanSN/SwiftBot/>, and paste it under the (catkin directory) in your system root folder.
3. Now, you must build the files. This can be done by first opening the terminal, changing the (working directory) to (catkin\_ws), and then typing the command (catkin\_make).
4. To install the Google Cartographer, the mapping algorithm, follow this link: <https://google-cartographer-ros.readthedocs.io/en/latest/compilation.html#building-installation>.
5. Now, you must source the files. This can be done by first opening the terminal, changing the (working directory) to (catkin\_ws), and then typing the command (source devel/setup.bash).
6. To run the mapping algorithm, you need to upload the (teleop code) under the (Arduino directory), on github, to Arduino. You also need to install the (teleop\_twist\_keyboard) package, by following this link: [http://wiki.ros.org/teleop\\_twist\\_keyboard](http://wiki.ros.org/teleop_twist_keyboard)

keyboard. ssh to the Pi, using your personal system, making sure that it is connected to the same WiFi.

7. Open a terminal on your system, and type in (roscore), to turn on ROS on your personal system. After that, using the ssh(ed) terminals, change the (ROS\_MASTER\_URI), and (ROS\_IP). This is to be done so that the robot can communicate its data with your system. You must change these two variables, for each terminal you ssh into the robot(pi), and also each terminal you open in your PC, ensuring two way communication. This can be done as follows:

For your PC:

- (a) (export ROS\_MASTER\_URI= http://[Your PC local IP address]:11311).
- (b) (export ROS\_IP= [Your PC local IP address]).

For your robot(pi):

- (a) (export ROS\_MASTER\_URI= http://[Your PC local IP address]:11311).
- (b) (export ROS\_IP= [robot(pi) local IP address]).

8. Now, for every command, open a terminal, and run them in parallel. Also, every one of these terminals need to have (catkin\_ws) as the (working directory).

Following codes need to be run on the ssh terminals:

- (a) (roslaunch rplidar\_ros rplidar.launch) (This will turn on the LIDAR).
- (b) (roslaunch teleop\_twist\_keyboard teleop\_twist\_keyboard.py).
- (c) (roslaunch motor\_driver motor\_driver.py).
- (d) (source install\_isolated/setup.bash).
- (e) (roslaunch cartographer\_ros cartographer.launch).

This code must be run on your local machine (PC):

- (a) (rviz rviz).



Once that is done, go to terminal where you ran the (teleop) command, and then follow the instructions to move the robot. You will notice the map being made on rviz. (Note: go to add topics tab in rviz, and click on the topics that you want see on the GUI. Particularly the map topic would be of use here).

9. If you need to run the main program, then you need to make sure that your Pi can be port forwarded into. For our system, we used ngrok on our Pi to make the server available for the app to access, this allowed us to skip the whole port forwarding process. However, this also means that we need to change the IP every time on our server when the robot starts (A limitation unless we buy the paid version of ngrok). This is needed, so that the app can open the lock of the robot using the pin security feature. The challenge is how to download ngrok on your Pi, fear not, we will still cover that here:

- (a) ssh into your Pi, and type into your terminal (wget) "<https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-arm.zip>".
- (b) Then unzip it by typing into your terminal (unzip /path/to/ngrok.zip).
- (c) (./ngrok authtoken <YOUR\_AUTH\_TOKEN>).
- (d) Finally (./ngrok http 4010).

10. Finally, moving onto the Navigation Stack (main program), you need to upload the main code under the (Arduino directory), on github, to Arduino. Now, for every command, open a terminal, and run them in parallel (Again keeping in mind step). Also, every one of these terminals need to have (catkin\_ws) as the (working directory).

These codes need to be run on the ssh terminals:

- (a) (roslaunch rplidar\_ros rplidar.launch) (This will turn on the LIDAR).
- (b) (roslaunch my\_robot\_name\_2dnav my\_robot\_configuration.launch).

These codes need to be run on your local machine:

- (a) `(roslaunch my_robot_name_2dnav move_base.launch)`.
- (b) `(rviz rviz)` (You need to set the initial pose of the robot, you can also use it to visualize where your robot is moving. You may close it only after setting the initial pose of the robot, this needs to be done once only).
- (c) `(python src/simple_navigation_goals/src/call_this.py)`.

Again, move to one of your ssh terminals, and type in:

- (a) `(python src/simple_navigation_goals/src/open_lock.py)`.
- (b) `(python src/simple_navigation_goals/src/mainLoop_f.py)`.

11. Else, you can just use the robot's pose using the (2d\_pose) icon on top, and then give a goal by pressing on the (2d\_navigation goal) icon on rviz, if you do not want to use the app, and just watch the robot move (smoothly) across the map, using your own coordinates. Just make sure that you don't input anything after the rviz command, ignoring every command that is written in point 11 after the rviz command, to be specific.

(Note: Go to add topics tab in rviz, and click on the topics that you want see on the GUI. Particularly the map, polygon, the global path, and costmap. The local path and costmap would be of use here).

## APPENDIX C

### FACIAL RECOGNITION TUTORIAL

#### **Files Related to the Facial Recognition Setup on the Server Side**

1. Make sure that (Python-Pip) is installed on the machine. This tutorial can be followed on a Windows machine with a valid Python installation: <https://www.liquidweb.com/kb/install-pip-windows/>.
2. Using (pip) in the cmd environment, further dependency libraries need to be installed which are mentioned in the following link:  
<https://packaging.python.org/tutorials/installing-packages/>.
  - (a) Scikit-learn (pip install scikit-learn).
  - (b) NumPy (pip install numpy).
  - (c) OpenCV (pip install cv2).
  - (d) Flask (pip install flask).
3. Now, to overcome any discrepancy in the trained data set, the module should be retrained over the current dependencies.

For this step, open a cmd prompt, and run (python extract\_embeddings.py), afterwards run (python train\_model.py).
4. After these two files are successfully executed, the folder (./dataset) will have folders of people who are recognizable by the system.

After this, the server should be run on the local device, the port number is customizable in the script (Server.py).
5. Installing ngrok (For port forwarding of the server running on localhost)

Setting up on a windows machine:

- (a) Download the ngrok ZIP file.
- (b) Unzip the ngrok.exe file.
- (c) Place the ngrok.exe in a folder of your choosing.
- (d) Make sure the folder is in your PATH environment variable.

For Linux machines:

The ngrok can be installed via terminal (`sudo apt-get install ngrok`) on Debian based systems while, (`sudo pacman -S ngrok`) on arch based systems directly.

- 6. After running the ngrok on desired port, the port forwarding will start, and the generated link would have to be updated in the (`src/simple_navigation_goals/mainloop.py`) file, in the (`Send_Nodes`) method.
- 7. After the server has been successfully set up, the robot can start communicating with the server to successfully run the facial recognition feature for security purposes.

## APPENDIX D

### ANDROID APPLICATION TUTORIAL

#### Files Related to Android App

Following are the steps that you need to follow in order to get the app in the working condition:

1. Download NPM, ie Node Package Manager.
2. Open cmd, and type (npm install ionic).
3. Now navigate to the app folder with the source code inside the terminal, and type (npm install). Now, npm will start installing all the dependencies.
4. Once that is done, type (ionic serve) to start the app. This will open the app in your default browser.
5. For running app on your android phone, connect your android phone to your PC, and run the command (ionic cordova run android --device --livereload).
6. After you have tested, and run the code, you can get a (.apk) file for the project by running the command (ionic cordova build --release android), after a while, you can find your (.apk) file here: (\platforms\android\app\build\outputs\apk\release\app-release-unsigned.apk).

#### Signing Apk

1. (keytool -genkey -v -keystore my-release-key.keystore -alias alias\_name -keyalg RSA -keysize 2048 -validity 10000).

If (keytool) is not found, use:

2. ("C:\ProgramFiles\Java\jre1.8.0\\_151\bin\keytool.exe" -genkey -v -keystore my-release-key.keystore -alias alias\_name -keyalg RSA -keysize 2048 -validity 10000).

3. (.keystore) file has been generated. To attach it with the unsigned apk, use the (OutSign) software. Path to the JDK file: (C:\ProgramFiles\Java\jdk1.8.0\\_144\bin).

## APPENDIX E

### FIRST PROTOTYPE KINEMATICS SIMULATIONS ON MATLAB

The simulation in Figure E.1 was for the first prototype. A differential drive wheeled mobile robot, given the control inputs  $WR$  and  $WL$  – the right and left wheel angular velocities, this simulation provided the instantaneous pose of the robot. The dotted line is the trajectory traced via the continuous integration method, and the purple line is the trajectory traced via the geometrical model-1 integration method. The geometrical model-1 integration method is the most accurate of all the discrete time integration methods since, it is the closest to the continuous integration method. For this simulation  $WR = WL$ . Therefore, the robot moved in a straight line.

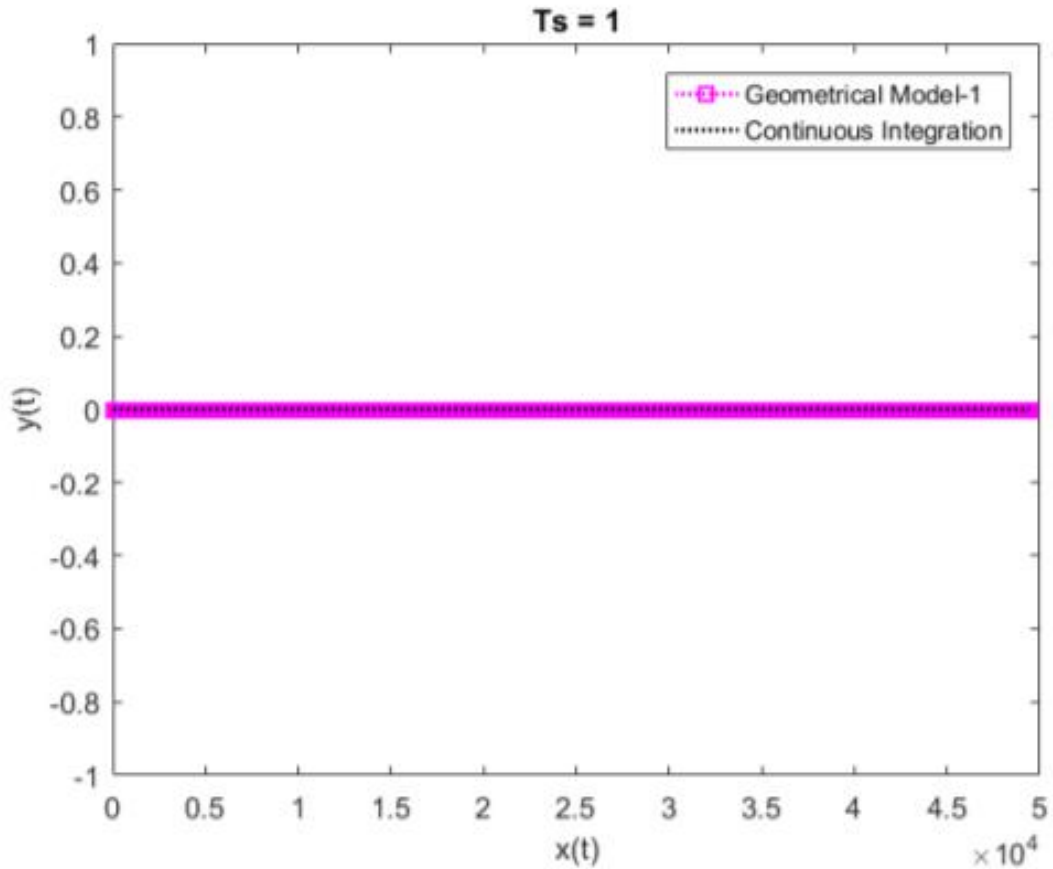


Figure E.1: Linear Trajectory

For the simulation in Figure E.2, WR was not kept equal to WL. Therefore, a circular trajectory was traced out, the robot made a turn.

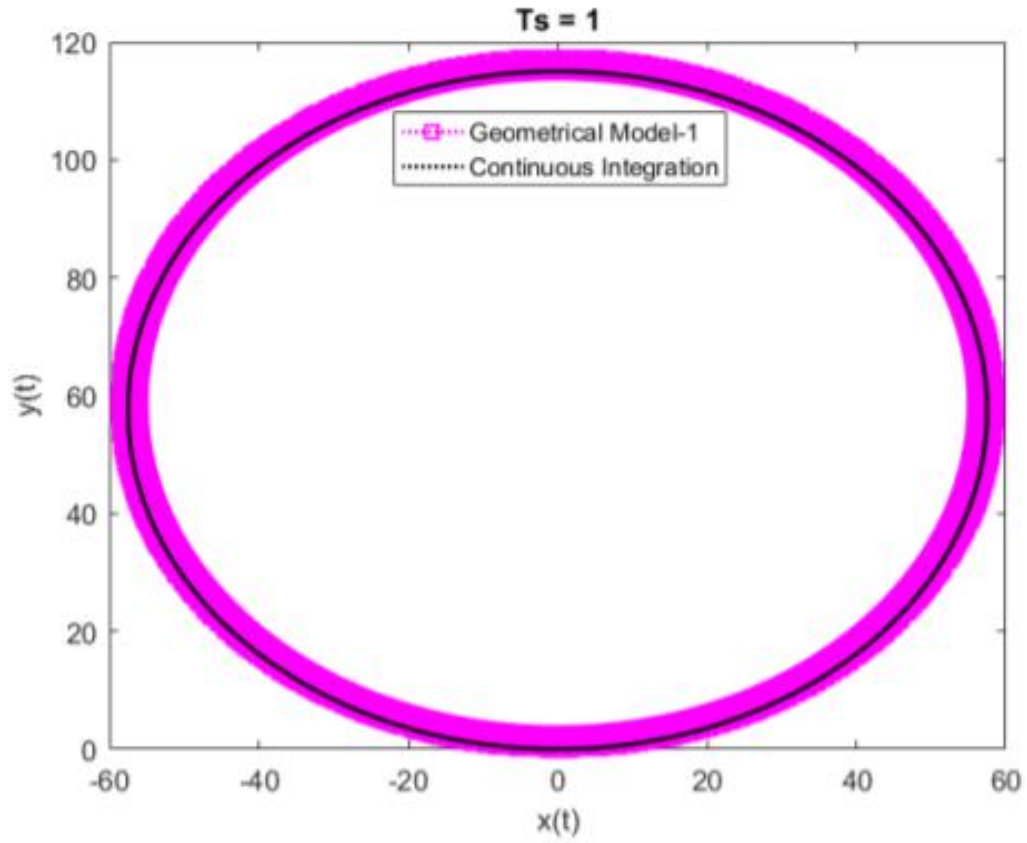


Figure E.2: Elliptical Trajectory

For the simulation in Figure E.3, a circular pose was provided and inverse kinematics was performed to determine the linear and angular velocities of the robot to allow it to follow the trajectory at run time.



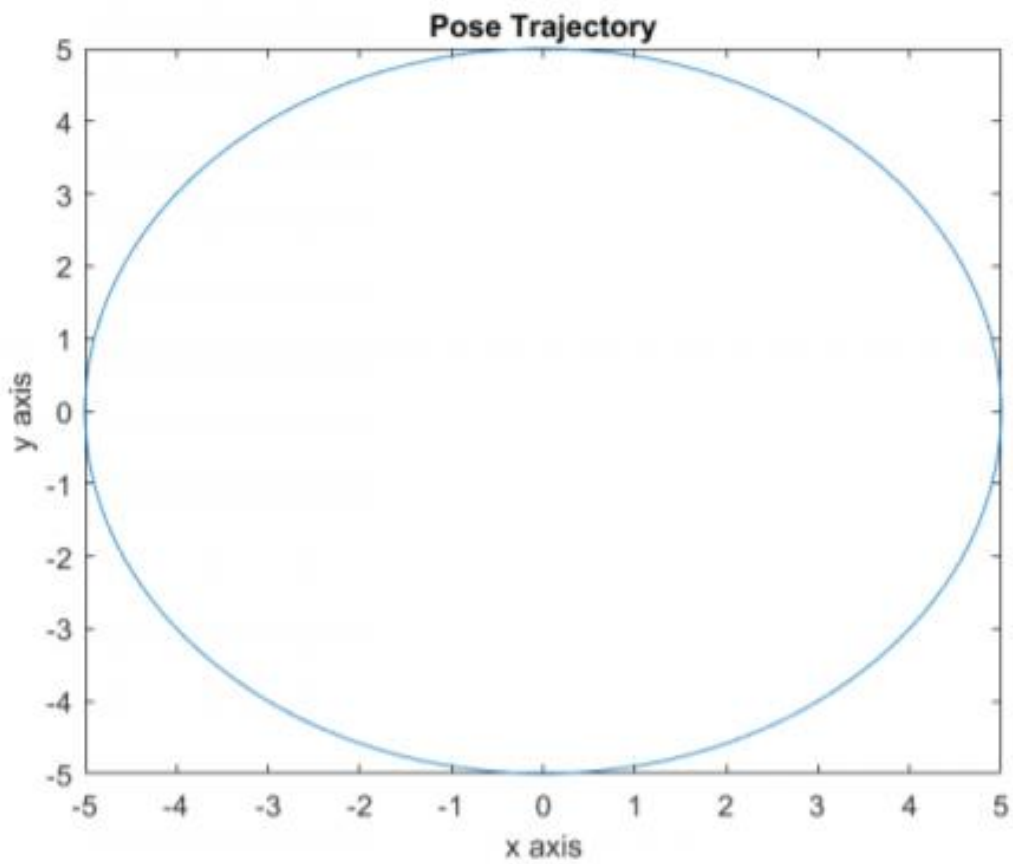


Figure E.3: Pose for Inverse Kinematics

Constant instantaneous linear and angular velocities were achieved for a circular trajectory at run time, as shown in Figure E.4.

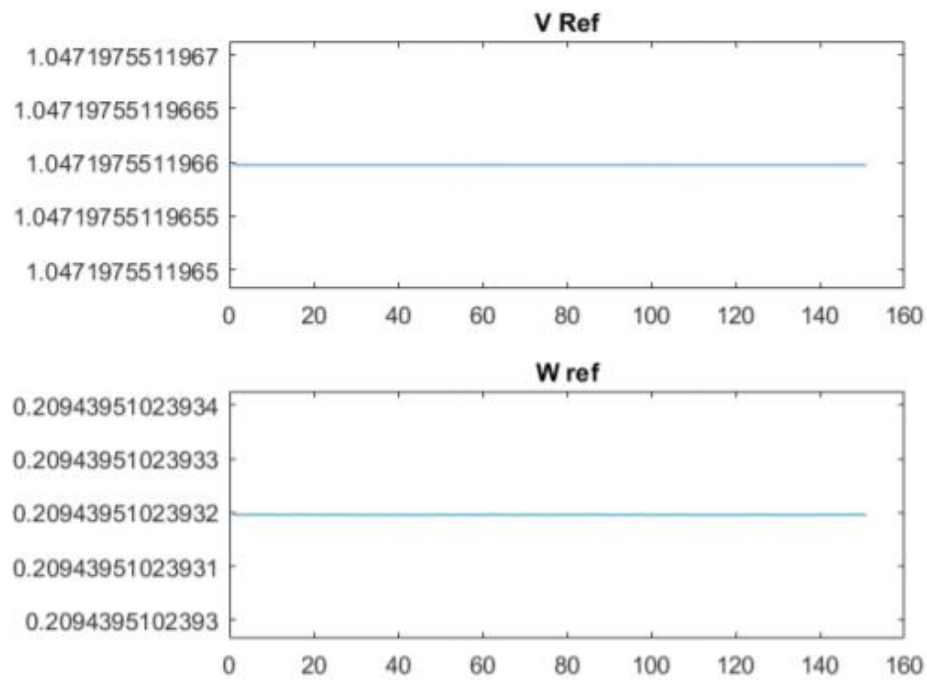


Figure E.4: Kinematic Control Inputs Given Pose

The simulation in Figure E.5 and Figure E.6 was of a bug 0 algorithm, the initial and final poses ( $x, y$ , and  $\Phi$ ) were provided and obstacles were inserted at specific coordinates. The robot was provided with the linear and angular velocities at run time, using inverse kinematics, to allow it to move from the initial pose to the final pose. When the robot encountered an obstacle, it followed it until it could head to the goal again.

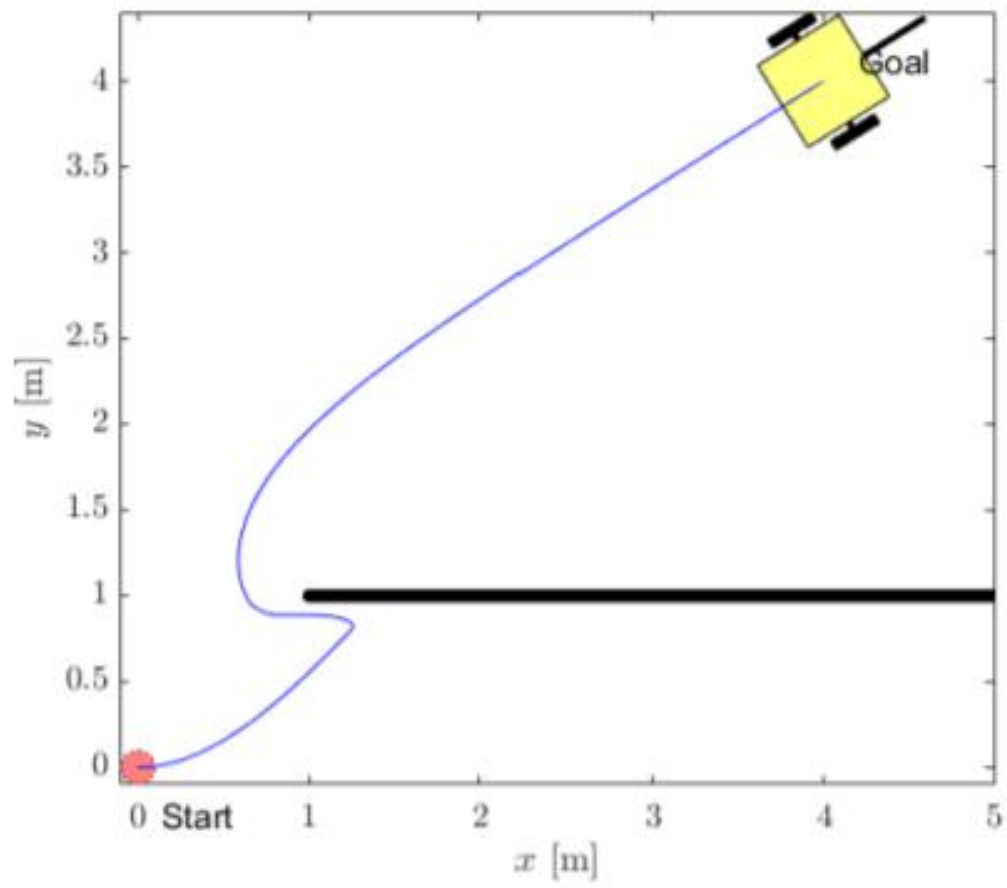


Figure E.5: Obstacle Avoidance and Go-to-Goal Behavior

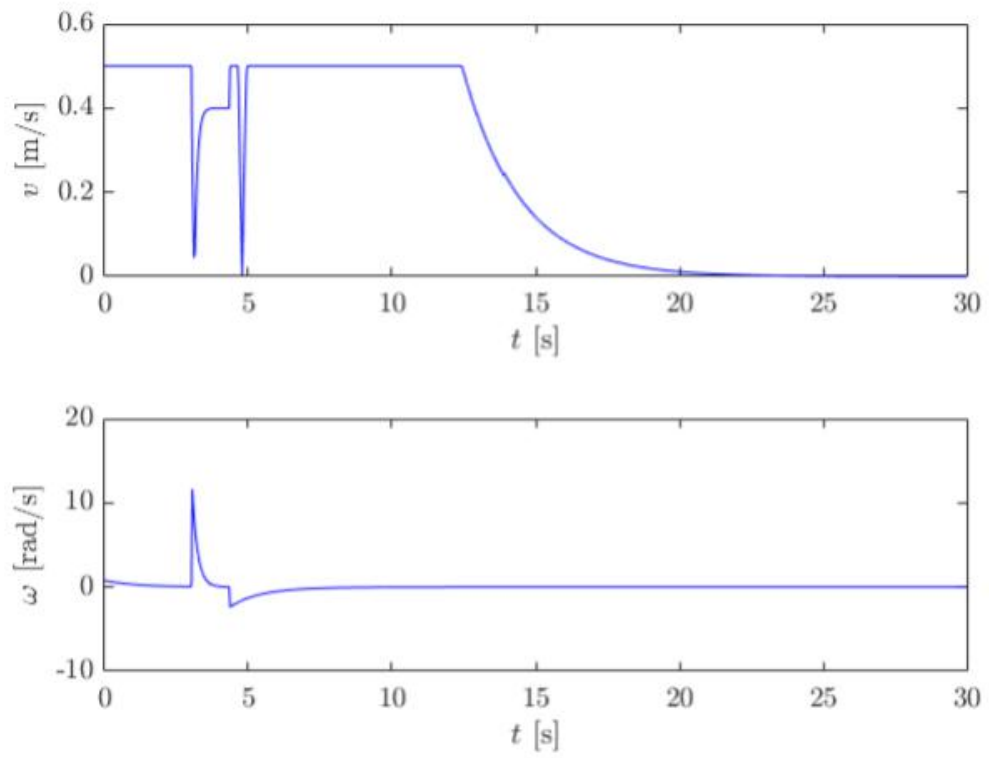


Figure E.6: Control Inputs Using Inverse Kinematics

## APPENDIX F

### ARDUINO CODE

#### F.1 Motor RPM Measurement Code

```
// RPM Measurement of Encoder Motors //
/*
*- encVal/pulse_per_rev revolutions in ----- interval
  ↳ /1000 seconds b/c interval given in milliseconds
*- ? revolutions in ----- 60 s
*- ? = (enc/pulse_per_rev)*60 / (int/1000) RPM i.e.
  ↳ revolutions in 60s
*/
int pulse_per_rev = 1120;
int interval = 50;
unsigned long current_t = 0;
float t = 60*(1000/interval);
float rpmA, rpmB;
int encoderPinA = 2;
int encoderPinB = 3;
volatile long encoderValA = 0;
volatile long encoderValB = 0;
int enA1 = 8;
int enA2 = 7;
int pwmA = 6;
int enB1 = 13;
int enB2 = 12;
int pwmB = 11;
```

```

void updateA() {
    encoderValA++;
}

void updateB() {
    encoderValB++;
}

void setup() {
    Serial.begin(9600);

    pinMode(encoderPinA, INPUT);
    pinMode(encoderPinB, INPUT);

    attachInterrupt(1, updateA, RISING);
    attachInterrupt(0, updateB, RISING);

    pinMode(enA1, OUTPUT);
    pinMode(enA2, OUTPUT);
    pinMode(pwmA, OUTPUT);
    pinMode(enB1, OUTPUT);
    pinMode(enB2, OUTPUT);
    pinMode(pwmB, OUTPUT);

    digitalWrite(enA1, HIGH);
    digitalWrite(enA2, LOW);
    digitalWrite(enB1, LOW);
    digitalWrite(enB2, HIGH);
}

void loop() {
    analogWrite(pwmA, 200);
    analogWrite(pwmB, 200);

    if (millis() >= current_t + interval) {
        detachInterrupt;
        current_t += interval;
    }
}

```

```

    rpmA = encoderValA*(t/pulse_per_rev);
    rpmB = encoderValB*(t/pulse_per_rev);
    encoderValA = 0;
    encoderValB = 0;

    Serial.println(rpmA); Serial.print("_");
    Serial.print(rpmB); Serial.print("_");

    attachInterrupt;

}

}

```

## F.2 Arduino Based Proportional-Integral Controller

```

// Motor encoder output pulse per rotation (change as
  ↪ required)
double enc_count_rev = 1120; //16*70 CPR

// Encoder output to Arduino Interrupt pin
#define ENCA 2
#define ENCB 3

// PWM/Speed pins for Motors
#define ENA 9
#define ENB 6

// Motor Directions
#define IN1A 10
#define IN2A 11
#define IN1B 7
#define IN2B 8

```

```

// PID constants
double ki = 0.7;
double kp = 0.5;
float error = 0;
long cumerror = 0;
long out = 0;
double gain = 1.1; //have to set this by comparing motors

// RPM calculation
long currentMillis;
long prevMillis = 0;
double interval = 10; //ms
double left_rpm, right_rpm;
double elapsedTime;
volatile long encoderValueA;
volatile long encoderValueB;

void setup() {
    // Setup Serial Monitor
    Serial.begin(57600);

    // Set PWM and DIR connections as outputs
    pinMode(ENA, OUTPUT);
    pinMode(IN1A, OUTPUT);
    pinMode(IN2A, OUTPUT);
    pinMode(ENB, OUTPUT);
    pinMode(IN1B, OUTPUT);
    pinMode(IN2B, OUTPUT);

```



```

// Attach interrupt
attachInterrupt(digitalPinToInterrupt(ENCA),
    ↪ updateEncoderA, RISING);
attachInterrupt(digitalPinToInterrupt(ENCB),
    ↪ updateEncoderB, RISING);

// Move motor forward
digitalWrite(IN1A, HIGH);
digitalWrite(IN2A, LOW);
analogWrite(ENA, 100);
digitalWrite(IN1B, LOW);
digitalWrite(IN2B, HIGH);
analogWrite(ENB, 120);
}

void loop() {
    currentMillis = millis();
    elapsedTime = currentMillis - prevMillis;
    if(elapsedTime > interval) {
        noInterrupts();
        prevMillis = millis();
        right_rpm = (encoderValueA/enc_count_rev)*60*1000;
        right_rpm = right_rpm/interval;
        left_rpm = (encoderValueB/enc_count_rev)*60*1000;
        left_rpm = left_rpm/interval;
        encoderValueA = 0;
        encoderValueB = 0;
        interrupts();
    }
}

```

```

    }

    error = right_rpm - left_rpm;
    cumerror += error*elapsedTime/1000;
    out = kp*error+ ki*cumerror;

    if (error >=0) {
        left_rpm = left_rpm + abs(out)*gain;
        left_rpm = left_rpm*(255/150);
        analogWrite(ENB, left_rpm);
        // delay(10);
    }
    else {
        left_rpm = left_rpm - abs(out)*gain;
        left_rpm = left_rpm*(255/150);
        analogWrite(ENB, left_rpm);
        // delay(10);
    }

    Serial.print(error); Serial.println("_error");
    Serial.print(left_rpm); Serial.println("_left_rpm");
    Serial.print(right_rpm); Serial.println("_right_rpm");
    Serial.println("_");
    delay(10);
}

void updateEncoderA() {
    // Increment value for each pulse from encoder
    encoderValueA++;
}

void updateEncoderB() {

```

```
// Increment value for each pulse from encoder
encoderValueB++;
}
```

### F.3 Arduinio Code Interfaced with ROS

```
# include <Encoder.h>
# include <ros.h>
# include <std_msgs/Int16.h>
# include <geometry_msgs/Twist.h>
# include <std_msgs/Float32.h>
# include <std_msgs/String.h>
# include <TimerOne.h>

Encoder knobLeft(2,4) ;
Encoder knobRight(3,5);

#define LOOP_TIME 100000
#define ENA 6
#define ENB 9
#define IN1 24
#define IN2 26
#define IN3 7
#define IN4 8
int relay_pin= 22;
float left_pwm, right_pwm;

ros :: NodeHandle nh;
std_msgs::Int16 left_wheel_enc;
```

```

std_msgs::Int16 right_wheel_enc;
std_msgs::Int16 right_wheel_values;
std_msgs::Int16 left_wheel_values;
std_msgs::Float32 open_lock_value;
std_msgs::Float32 check_open_lock;

ros :: Publisher check_open_lock_pub("open_lock", &
    ↪ check_open_lock);
ros :: Publisher left_wheel_enc_pub("left_encoder", &
    ↪ left_wheel_enc);
ros :: Publisher right_wheel_enc_pub("right_encoder", &
    ↪ right_wheel_enc);
void timerIsr()
{
    Timer1.detachInterrupt(); //stop the timer
    right_wheel_enc.data=knobRight.read();
    left_wheel_enc.data=knobLeft.read();
    right_wheel_enc_pub.publish(&right_wheel_enc);
    left_wheel_enc_pub.publish(&left_wheel_enc);
    Timer1.attachInterrupt(timerIsr); //enable the timer
}

void cmdLeftWheelCB(const std_msgs::Float32& msg)
{
    left_wheel_values.data = msg.data;
}

void cmdRightWheelCB(const std_msgs::Float32& msg)
{

```

```

    right_wheel_values.data = msg.data;
}

void open_lock(const std_msgs::Float32& msg)
{
    if(msg.data > 0){
        check_open_lock.data=msg.data;
        check_open_lock_pub.publish(&check_open_lock);
        digitalWrite(relay_pin, HIGH);
        delay(5000);
        check_open_lock.data=0;
        check_open_lock_pub.publish(&check_open_lock);
        digitalWrite(relay_pin, LOW);
    }
}

ros::Subscriber<std_msgs::Float32> subCmdLeft("left_pwm"
    ↪ , &cmdLeftWheelCB);
ros::Subscriber<std_msgs::Float32> subCmdRight("right_pwm"
    ↪ ", &cmdRightWheelCB);
ros::Subscriber<std_msgs::Float32> openLock("locker", &
    ↪ open_lock);

void setup() {
    // put your setup code here, to run once:
    Serial.begin(57600);
    pinMode (ENA, OUTPUT) ;
    pinMode (ENB, OUTPUT) ;
    pinMode ( IN1 , OUTPUT) ;

```

```

pinMode ( IN2 , OUTPUT) ;
pinMode ( IN3 , OUTPUT) ;
pinMode ( IN4 , OUTPUT) ;
pinMode ( relay_pin, OUTPUT) ;
analogWrite(ENA, 0);
analogWrite(ENB, 0);
digitalWrite(IN1, LOW);
digitalWrite(IN2, LOW);
digitalWrite(IN3, LOW);
digitalWrite(IN4, LOW);
Timer1.initialize(LOOP_TIME);
nh.initNode();
nh.subscribe(subCmdLeft);
nh.subscribe(subCmdRight);
nh.subscribe(openLock);
nh.advertise(left_wheel_enc_pub);
nh.advertise(right_wheel_enc_pub);
nh.advertise(check_open_lock_pub);

Timer1.attachInterrupt(timerIsr);
left_wheel_values.data=0;
right_wheel_values.data=0;
}

void loop() {

nh.spinOnce();

```

```

if (left_wheel_values.data > 0 && right_wheel_values.
    ↪ data > 0) //FWD Condition
{
    digitalWrite(IN1,HIGH); //right forward
    digitalWrite(IN2,LOW);
    digitalWrite(IN3,HIGH); //left forward
    digitalWrite(IN4,LOW);
}
else if (left_wheel_values.data < 0 &&
    ↪ right_wheel_values.data < 0)
{
    digitalWrite(IN1,LOW); //right backwards
    digitalWrite(IN2,HIGH);
    digitalWrite(IN3,LOW); //left backwards
    digitalWrite(IN4,HIGH);
}

else if (left_wheel_values.data > 0 &&
    ↪ right_wheel_values.data < 0)
{
    digitalWrite(IN1,LOW); //right backwards
    digitalWrite(IN2,HIGH);
    digitalWrite(IN3,HIGH); //left forward
    digitalWrite(IN4,LOW);
}

else if (left_wheel_values.data == 0 &&
    ↪ right_wheel_values.data == 0)
{

```

```
    digitalWrite(IN3, LOW); //left backward
    digitalWrite(IN4, LOW);
    digitalWrite(IN1, LOW); //right forward
    digitalWrite(IN2, LOW);
}

else
{
    digitalWrite(IN3, LOW); //left backward
    digitalWrite(IN4, HIGH);
    digitalWrite(IN1, HIGH); //right forward
    digitalWrite(IN2, LOW);
}

    analogWrite(ENB, abs(left_wheel_values.data));
    analogWrite(ENA, abs(right_wheel_values.data));
}
```



## APPENDIX G

### PYTHON SCRIPTS

#### G.1 Python Script for Data Formatting

```
import csv
file = open("putty4.txt", "r")
#list of lines after ignoring initial text
#and final incomplete values(if any)
lines = file.read().splitlines()[2:-2]
#list of data points
lst = []
#splitting data into list of lists based on empty space
for data in lines:
    lst.append(data.split())
colA = []
colB = []
for i in lst:
    colA.append(i[0])
    colB.append(i[1])

with open('MotorSpeeds2.csv', 'w', newline='') as myfile:
    wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
    for word in lst:
        wr.writerow(word)
```

#### G.2 Main Loop

```
#!/usr/bin/env python
```

```

import requests
import json, os
import cv2
import rospy
from std_msgs.msg import Float32
import random
import time
from new import funct #uncomment this when on pi

headers={'content-type' : 'image/jpg'}

currentBookingInfo = ""
currentLocation = [1,1]
customerId = None

pub = rospy.Publisher('locker', Float32, queue_size=10)
rospy.init_node('talker', anonymous=False)

URL='https://186ad9c5edee.ngrok.io/upload'

def send_nodes(img_file=None):
    f=open("Gray.jpg","rb")
    img={'file': f}
    response=requests.post(URL,files=img)
    response=response.json()
    print(response)
    return response['message']

def getNextBookingDetails(currentBookingId):

```

```

URL = "https://api.anomoz.com/api/swift/post/
    ↪ read_new_booking.php?currentBookingId="+str(
    ↪ currentBookingId)
r = requests.get(url = URL)
# extracting data in json format
data = r.json()
info = (data[0])
return info

def markBookingAsEnded(currentBookingId):
    URL = "https://api.anomoz.com/api/swift/post/
        ↪ mark_booking_as_ended.php?currentBookingId="+str(
        ↪ currentBookingId)
    r = requests.get(url = URL)
    return True

def getPickupLocation(currentBookingInfo):
    print("getPickupLocation_set:_", [currentBookingInfo["
        ↪ fromLocation_lat"], currentBookingInfo["
        ↪ fromLocation_lng"]])
    return([currentBookingInfo["fromLocation_lat"],
        ↪ currentBookingInfo["fromLocation_lng"]])

def getFinalLocation(currentBookingInfo):
    print("getFinalLocation_set:_", [currentBookingInfo["
        ↪ toLocation_lat"], currentBookingInfo["
        ↪ toLocation_lng"]])
    return([currentBookingInfo["toLocation_lat"],
        ↪ currentBookingInfo["toLocation_lng"]])

```

```

def startNavigator(currentLocation, destination):
    #programCommand = "roslaunch simple_navigation_goals
        ↪ simple_navigation_goals _param_x:="+str(
        ↪ destination[0])+ " _param_y:="+str(destination[1])
    #if os.system(programCommand) == 0:
        #print("program finished successfully")
    #else:
        #print("launch failed")

def startNavigator(resp):
    # programCommand = "roslaunch simple_navigation_goals
        ↪ simple_navigation_goals _param_x:="+str(
        ↪ destination[0])+ " _param_y:="+str(destination[1])
    # if os.system(programCommand) == 0:
    # print("program finished successfully")
    # else:
    # print("launch failed")
    URL = "http://192.168.86.28:2015/movebase?resp="+str(
        ↪ resp[0])+ "&resp1="+str(resp[1])
    r = requests.get(url = URL)
    return True

def generateExe():
    if os.system("g++ -g foo.cpp -o foo") == 0:
        print ("exe_compiled")
    else:
        print ("exe_compiling_Failed")

```

```

def open_lock(data):
    pub.publish(data)

    # try:
        #rospy.init_node('talker', anonymous=True)
    #except:
        #print("some")

    print("hello")

    rospy.Subscriber("open_lock", Float32, callback)

    rospy.sleep(5.0)

def callback(data):
    print(data.data, "this_is_data")

    # time.sleep(5.0)
    # if data.data=="open":
    # time.sleep(10.0)
    # pub.publish("close")
    # time.sleep(5.0)
    # if data.data=="close":
    # rospy.sleep(10.0)
    # # return "some"
    # # rospy.signal_shutdown("The lid has been closed")
    # # return "some"

def Get_pin():
    #send the number as a callback to customer to enter
    ↪ pin

    #get callback from app that the pin has been entered
    ↪ successfully

    try:

```

```

        file1 = open("lockstatus.txt", "r+")
        fileinp = file1.read()
        file1.close()
        if (fileinp=="opened"):
            return True
        else:
            return False
    except:
        return False

def runSingleCycle():

    currentBookingInfo = getNextBookingDetails(0)
    #currentBookingInfo = {'bookingId': '187', 'timeAdded
    ↪ ': '1573634449', 'fromLocation': 'Auditorium', '
    ↪ toLocation': 'Auditorium', 'fromPerson': 'Dr. taj
    ↪ ', 'toPerson': 'Ahsan', 'status': 'waiting', '
    ↪ fromLocation_lat': '1.17', 'fromLocation_lng':
    ↪ '0.77', 'toLocation_lat': '0.17', 'toLocation_lng
    ↪ ': '-0.775'}

    print("currentBookingInfo_updated:_",
    ↪ currentBookingInfo)

    customerId=currentBookingInfo['toPerson']
    #go to pickup location
    pickupLocation = getPickupLocation(currentBookingInfo)
    print(startNavigator(pickupLocation)) #program stops
    ↪ until

    #print (startNavigator(currentBookingInfo))
    #open lock for pickup

```

```

open_lock(1)

#to to destination
dropoffLocation = getFinalLocation(currentBookingInfo)
print("dropoffLocation", dropoffLocation)
startNavigator(dropoffLocation)

#openlock for dropoff
name=""

if (Get_pin()==False):
    try:
        name=send_nodes(funct())
    except:
        print("no")
if name==customerId:
    open_lock(1)
    print("Need_pin_to_unlock")
else:
    while (True):
        if (Get_pin()==True):
            open("lockstatus.txt", 'w').close()
            break
open_lock(1)
#mark booking
markBookingAsEnded(currentBookingInfo['bookingId'])

#runSingleCycle()

```

```

def main():

    #generateExe()

    runSingleCycle()

    #verifyUser()

    #pass


    #get next booking Status
# print(send_nodes(funcnt())) #uncomment this when on pi
main()
rospy.signal_shutdown("The_lid_has_been_closed")
#print()

```

### G.3 open\_lock.py

```

from flask import Flask
app = Flask(__name__)

from flask import request

@app.route('/openlock')

def openlock():

    isAuth = request.args.get('auth')

    print("inp", isAuth)

    if(isAuth=="true"):

        file1 = open("lockstatus.txt","w")

        file1.write("opened")

```



```

        file1.close()

        #open_lock("open")

        return "true"

    else:

        return "false"

if __name__ == '__main__':

    app.run(host= '0.0.0.0',port=4010)

```

#### G.4 call\_this.py

```

from flask import Flask
app = Flask(__name__)
from flask import request
import os

def getPickupLocation(currentBookingInfo):
    print("getPickupLocation_set:_", [currentBookingInfo["
        ↪ fromLocation_lat"], currentBookingInfo["
        ↪ fromLocation_lng"]])
    return([currentBookingInfo["fromLocation_lat"],
        ↪ currentBookingInfo["fromLocation_lng"]])

def getFinalLocation(currentBookingInfo):
    print("getFinalLocation_set:_", [currentBookingInfo["
        ↪ toLocation_lat"], currentBookingInfo["
        ↪ toLocation_lng"]])

```

```

    return ([currentBookingInfo["toLocation_lat"],
            ↪ currentBookingInfo["toLocation_lng"]])

def startNavigator(destination):
    programCommand = "roslaunch simple_navigation_goals_
        ↪ simple_navigation_goals_ __param_x:="+str(
        ↪ destination[0])+ " __param_y:="+str(destination[1])
    if os.system(programCommand) == 0:
        return "True"
    else:
        return "False"

@app.route('/movebase')
def movebase():
    isAuth = request.args.get('resp')
    isAuth2 = request.args.get('resp1')
    var=[float(isAuth), float(isAuth2)]
    print(var)
    # isAuth= request.args.get('resp2')
    # pickupLocation = getPickupLocation(isAuth)
    # dropoffLocation = getFinalLocation(isAuth)

    resp = startNavigator(var)

    # return resp
    return resp

if __name__ == '__main__':

```

```
app.run(host="0.0.0.0",port=2015)
```

## APPENDIX H

### ROS LAUNCH AND CONFIGURATION FILES

#### H.1 base\_local\_planner\_params.yaml

```
TrajectoryPlannerROS:
  max_vel_x: 0.4
  min_vel_x: 0.2
  max_rotational_vel: 0.5
  max_vel_theta: 1.5
  min_vel_theta: -1.5
  min_in_place_rotational_vel: 0.25
  min_in_place_vel_theta: 0.7
  escape_vel: -0.25
  acc_lim_theta: 0.3
  acc_lim_x: 0.5
  acc_lim_y: 0.07
  holonomic_robot: false
  meter_scoring: true
  xy_goal_tolerance: 0.15
  yaw_goal_tolerance: 1.7
```

#### H.2 costmap\_common\_params.yaml

```
obstacle_range: 1.4
raytrace_range: 2.0

max_obstacle_height: 0.2
```

```
min_obstacle_height: 0.05

robot_radius: 0.20
inflation_radius: 0.55

transform_tolerance: 5.0
map_type: costmap
cost_scaling_factor: 100

map_topic: /map
subscribe_to_updates: true

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: laser, data_type:
    ↪ LaserScan, topic: scan , marking: true, clearing:
    ↪ true}
```

### H.3 global\_costmap\_params.yaml

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  rolling_window: false
  static_map: true
  publish_frequency: 1.0
  width: 15
```

```
height: 15
resolution: 0.1
```

#### **H.4 local\_costmap\_params.yaml**

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 2
  height: 2
  resolution: 0.025
```

#### **H.5 trajectory\_planner.yaml**

```
TrajectoryPlannerROS:
  max_vel_x: 0.4
  min_vel_x: 0.2
  max_rotational_vel: 0.5
  max_vel_theta: 1.5
  min_vel_theta: -1.5
  min_in_place_rotational_vel: 0.25
  min_in_place_vel_theta: 0.7
  escape_vel: -0.25
  acc_lim_theta: 0.3
  acc_lim_x: 0.5
```

```

acc_lim_Y: 0.07
holonomic_robot: false
meter_scoring: true
xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.3
latch_xy_goal_tolerance: true

```

## H.6 my\_robot\_configuration.launch

```

<launch>
<arg name="map_file" default=" /home/ubuntu/sameer_house.
  ↳ yaml"/>
<node name="map_server" pkg="map_server" type="map_server
  ↳ " args="$(arg map_file)">
  <param name="frame_id" type="str" value="map"/>
</node>
<include file="$(find amcl)/examples/amcl_diff.launch"/>
<rosparam param="ticks_meter">17825</rosparam>
<node pkg="differential_drive" type="diff_tf.py" name="
  ↳ odometry" output="screen">
  <remap from="lwheel" to="left_encoder" />
  <remap from="rwheel" to="right_encoder" />
  <rosparam param="base_width">0.40</rosparam>
  <rosparam param="odom_frame_id" subst_value="True"
    ↳ > "/odom" </rosparam>
  <rosparam param="base_frame_id" subst_value="True">
    ↳ "/base_link" </rosparam>
  <rosparam param="global_frame_id" subst_value="True
    ↳ "> "/map" </rosparam>

```

```

        <rosparam param="rate">50</rosparam>
</node>
<node pkg="differential_drive" type="pid_velocity.py"
  ↪ name="lpid">
    <remap from="wheel" to="left_encoder"/>
    <remap from="motor_cmd" to="left_pwm"/>
    <remap from="wheel_vtarget" to="left_target"/>
    <remap from="wheel_vel" to="left_actual"/>
    <rosparam param="Kp">400</rosparam>
    <rosparam param="Ki">10</rosparam>
    <rosparam param="Kd">1</rosparam>
    <rosparam param="out_min">-255</rosparam >
    <rosparam param="out_max">255</rosparam >
    <rosparam param="rate">40</rosparam >
    <rosparam param="timeout_ticks">4</rosparam>
    <rosparam param="rolling_pts">5</rosparam>
</node>

<node pkg="differential_drive" type="pid_velocity.py"
  ↪ name="rpid">
    <remap from="wheel" to="right_encoder"/>
    <remap from="motor_cmd" to="right_pwm"/>
    <remap from="wheel_vtarget" to="right_target"/>
    <remap from="wheel_vel" to="right_actual"/>
    <rosparam param="Kp">450</rosparam>
    <rosparam param="Ki">10</rosparam>
    <rosparam param="Kd">1</rosparam>
    <rosparam param="out_min">-255</rosparam>
    <rosparam param="out_max">255</rosparam>

```



```

        <rosparam param="rate">40</rosparam>
        <rosparam param="timeout_ticks">4</rosparam>
        <rosparam param="rolling_pts">5</rosparam>
</node>

<node pkg="differential_drive" type="twist_to_motors.py"
  ↳ name="twist" output="screen">
    <remap from="twist" to="cmd_vel"/>
    <remap from="lwheel_vtarget" to="left_target"/>
    <remap from="rwheel_vtarget" to="right_target"/>
    <rosparam param="base_width">0.40</rosparam>
</node>

<node pkg="roserial_python" type="serial_node.py" name="
  ↳ serial_node">
    <param name="port" value="/dev/ttyACM0"/>
    <param name="baud" value="57600"/>
</node>

<node pkg="tf" type="static_transform_publisher" name="
  ↳ laser_transform" args="0 0 0.14 4.71 0 0 base_link
  ↳ laser 100"/>
</launch>

```

## H.7 move\_base.launch

```

<launch>

  <master auto="start"/>

```

```

<node pkg="move_base" type="move_base" respawn="false"
  ↳ name="move_base" output="screen">
  <rosparam file="$(find my_robot_name_2dnav) /
    ↳ costmap_common_params.yaml" command="load" ns="
    ↳ global_costmap" />
  <rosparam file="$(find my_robot_name_2dnav) /
    ↳ costmap_common_params.yaml" command="load" ns="
    ↳ local_costmap" />
  <rosparam file="$(find my_robot_name_2dnav) /
    ↳ local_costmap_params.yaml" command="load" />
  <rosparam file="$(find my_robot_name_2dnav) /
    ↳ global_costmap_params.yaml" command="load" />
  <rosparam file="$(find my_robot_name_2dnav) /
    ↳ base_local_planner_params.yaml" command="load" />
  <param name="base_global_planner" type="string" value
    ↳ ="navfn/NavfnROS" />
  <param name="controller_frequency" type="double" value
    ↳ ="10.0"/>
  <param name="planner_frequency" type="double" value
    ↳ ="10.0"/>
</node>

</launch>

```

## H.8 simple\_navigation\_goals.cpp

```

#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

```

```

#include<iostream>

typedef actionlib::SimpleActionClient<move_base_msgs::
    ↪ MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv){
    ros::init(argc, argv, "simple_navigation_goals");

    std::float param_x;
    std::float param_y;

    ros::NodeHandle nh("~");

    nh.getParam("param_x",param_x);
    nh.getParam("param_y",param_y);
    std::cout << param_x;

    //tell the action client that we want to spin a thread
    ↪ by default
    MoveBaseClient ac("move_base", true);

    //wait for the action server to come up
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting_for_the_move_base_action_server_to_
            ↪ come_up");
    }

    move_base_msgs::MoveBaseGoal goal;

    //we'll send a goal to the robot to move 1 meter forward
    goal.target_pose.header.frame_id = "base_link";

```

```

goal.target_pose.header.stamp = ros::Time::now();

goal.target_pose.pose.position.x = param_x;
goal.target_pose.pose.position.y = param_y;
goal.target_pose.pose.orientation.w = 1;

ROS_INFO("Sending_goal");
ac.sendGoal(goal);

ac.waitForResult();

if(ac.getState() == actionlib::SimpleClientGoalState::
    ↪ SUCCEEDED)

    ROS_INFO("Hooray, the base moved 1 meter forward");
else

    ROS_INFO("The base failed to move forward 1 meter for
    ↪ some reason");

return 0;
}

```

**APPENDIX I**  
**2D LASER SCANNER GENERATED MAPS**



Figure I.1: Projects Lab Map 1

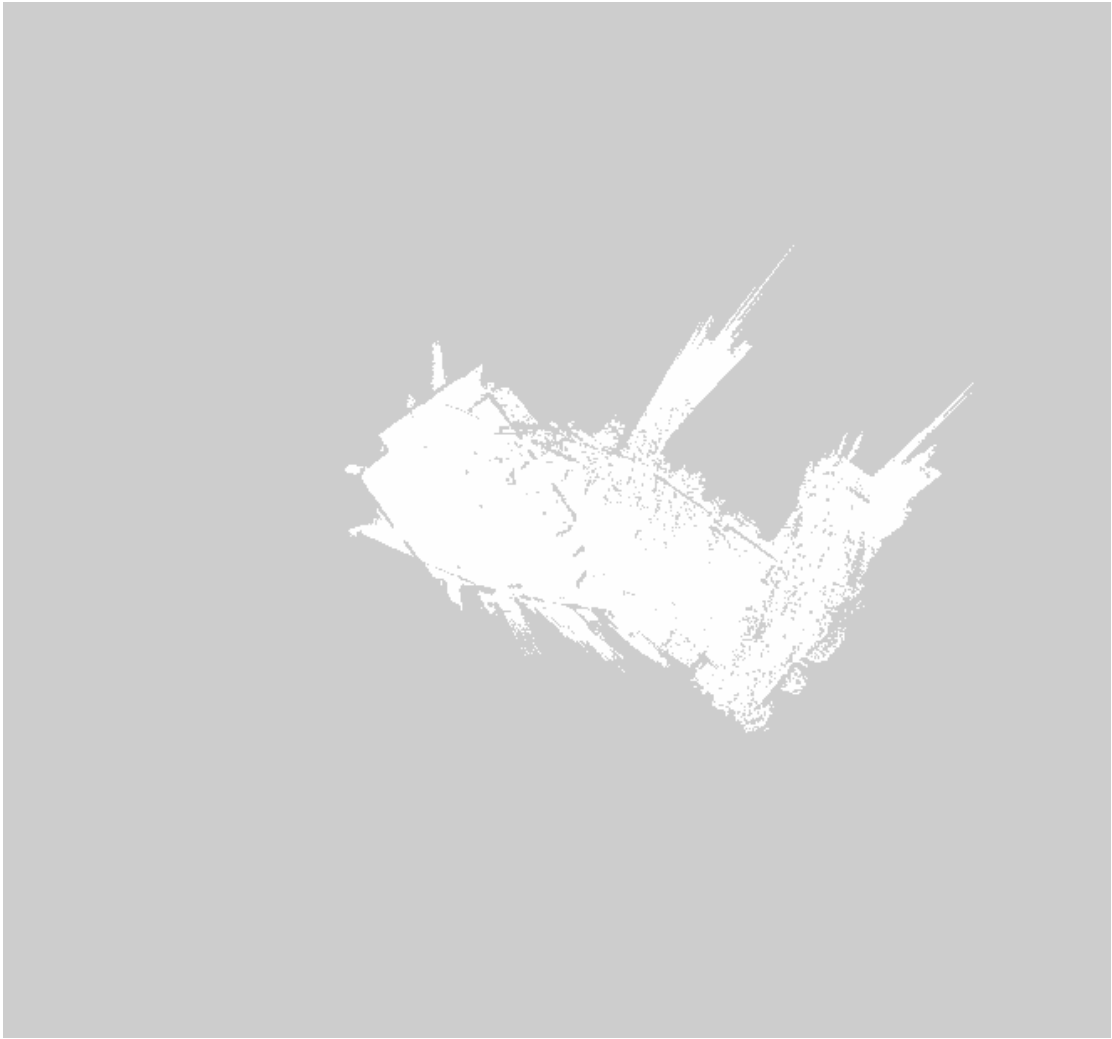


Figure I.2: Projects Lab Map 2



Figure I.3: House Map

Figure I.1 shows a map of a portion of the Projects Lab at Habib University, generated using the LIDAR mounted on Prototype 1 during early testing. As the map making process was refined by the group, more accurate maps came about. Figure I.2 shows another map of the same portion, as in Figure I.1, but with altered movements of the robot to better capture the area. Figure I.3 shows a map of a portion of the living room of one of the team members' house. This smaller map was developed in order to test the robot's movement with costmap parameters, set on a smaller map to observe straight line movement for the PID parameter refinement.

## REFERENCES

- [1] A. G. Özkil, Z. Fan, S. Dawids, H. Aanæs, J. K. Kristensen, and K. H. Christensen, "Service robots for hospitals: A case study of transportation tasks in a hospital," *IEEE International Conference on Automation and Logistics*, 2009, pp. 289–294.
- [2] M. D. Rossetti, A. Kumar, and R. A. Felder, "Mobile robot simulation of clinical laboratory deliveries," *Winter Simulation Conference*, 1998, pp. 1415–1422.
- [3] J. Evans, B. Krishnamurthy, W. Pong, R. Croston, C. Weiman, and G. Engelberger, "Helpmate: A Robotic Materials Transport System," *Robotics and Autonomous Systems*, pp. 251–256, 1989.
- [4] "Quarterly E-Commerce Report 1st Quarter 2018," U.S. Department of Commerce, Washington, D.C., Publication CB18-74, 2018.
- [5] K. Rogers, *Higher Demand for Quick Delivery Is Creating a Boom in Jobs*, <https://www.cnbc.com/2018/05/04/higher-demand-for-quick-delivery-is-creating-a-boom-in-jobs.html>, CNBC News.
- [6] C. Gartenberg, *Amazon is now delivering packages in Southern California with its Scout robots*, <http://www.theverge.com/2019/8/9/20798604/amazon-scout-robot-delivery-irving-southern-california-expansion-prime>, 2019.
- [7] A. J. Hawkins, *Thousands of autonomous delivery robots are about to descend on us college campuses*, <https://www.theverge.com/2019/8/20/20812184/starship-delivery-robot-expansion-college-campus>, 2019.
- [8] A. Weiner, *Minor Roadblocks Stand in the Way of Personal Delivery Devices*, <https://thespoon.tech/minor-roadblocks-stand-in-the-way-of-personal-deliverydevices/>, The Spoon, 2018.
- [9] M. Harris, *"our Streets Are Made for People : San Francisco Mulls Ban on Delivery Robots*, <https://www.theguardian.com/sustainable-business/2017/may/31/delivery-robots-drones-san-francisco-public-safety-job-lossfears-marble>, The Guardian, 2018.
- [10] A. Zaleski, *San Francisco to Delivery Robots: Get Off the Damn Sidewalk*, <https://www.citylab.com/life/2017/05/san-francisco-to-delivery-robots-get-off-the-damnsidewalk/527460/>, Citylab, 2018.



- [11] B. Zhang, *Personal Delivery Devices to Expand across Bay Area*, <https://thecampanile.org/2017/12/07/personal-delivery-devices-to-expand-across-bay-area/>, The Campanile, 2018.
- [12] E. Ackerman, *Sony Partners with CMU to Develop Food Prep and Delivery Robots*, <https://spectrum.ieee.org/automaton/robotics/home-robots/sony-partners-with-cmu-to-develop-food-prep-and-delivery-robots>, 2018.
- [13] T. S. Perry, *CES 2018: Delivery Robots Are Full-Time Employees at a Las Vegas Hotel*, <https://spectrum.ieee.org/view-from-the-valley/robotics/industrial-robots/ces-2018-delivery-robots-are-fulltime-employees-at-a-las-vegas-hotel>, 2019.
- [14] *Robotics/types of robots/wheeled*, [https://en.wikibooks.org/wiki/Robotics/Types\\_of\\_Robots/Wheeled](https://en.wikibooks.org/wiki/Robotics/Types_of_Robots/Wheeled), WIKIBOOKS, 2019.
- [15] *Robot mechanisms*, <https://rpal.cs.cornell.edu/foundations/mechanisms.pdf>, 2020.
- [16] *Pros and cons for different types of drive selection*, <https://robohub.org/pros-and-cons-for-different-types-of-drive-selection/>, Robohub, 2020.
- [17] R. Ron, *4 wheeled robot design basics and challenges*, <https://www.rakeshmondal.info/4-Wheel-Drive-Robot-Design>, RONRobotics, 2019.
- [18] *RPLIDAR A2 Introduction and Datasheet*, [https://www.generationrobots.com/media/robopeak\\_2d\\_lidar\\_brief\\_en\\_A2M4.pdf](https://www.generationrobots.com/media/robopeak_2d_lidar_brief_en_A2M4.pdf), SLAMTEC, 2020.
- [19] <https://www.pololu.com/product/2826>, Pololu, 2020.
- [20] <https://www.pololu.com/product/1451>, Pololu, 2020.
- [21] *Raspberry pi camera board v2.1 (8mp, 1080p)*, <https://uk.pi-supply.com/products/raspberry-pi-camera-board-v2-1-8mp-1080p>, PiSupply, 2020.
- [22] *Raspberry Pi 3 Model B*, <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>, Raspberry Pi Foundation, 2020.
- [23] M. Gudino, *Arduino uno vs. mega vs. micro*, <https://cutt.ly/OiYYU1B>, ARROW, 2020.
- [24] *Installation/ubuntu*, <http://wiki.ros.org/Installation/Ubuntu>, ROS.org, 2020.

- [25] *Installation/ubuntu*, <https://downloads.ubiquityrobotics.com/pi.html>, ROS.org, 2020.
- [26] *Catkin*, <http://wiki.ros.org/catkin>, ROS.org, 2020.
- [27] *Creating a workspace for catkin*, [http://wiki.ros.org/catkin/Tutorials/create\\_a\\_workspace](http://wiki.ros.org/catkin/Tutorials/create_a_workspace), ROS.org, 2020.
- [28] *Rosserial*, [http://wiki.ros.org/rosterial\\_python?distro=kinetic](http://wiki.ros.org/rosterial_python?distro=kinetic), ROS.org, 2020.
- [29] *Rplidar*, <http://wiki.ros.org/rplidar>, ROS.org, 2020.
- [30] *Navigation*, <http://wiki.ros.org/navigation>, ROS.org, 2020.
- [31] *Gmapping*, <http://wiki.ros.org/gmapping>, ROS.org, 2020.
- [32] *Openslam*, <https://openslam-org.github.io/gmapping.html>, ROS.org, 2020.
- [33] *Coordinate frames for mobile platforms*, <https://www.ros.org/repos/rep-0105.html>, ROS.org, 2020.
- [34] *Tf*, <http://wiki.ros.org/tf>, ROS.org, 2020.
- [35] <https://google-cartographer.readthedocs.io/en/latest/>, 2020.
- [36] H. Prasaath, *Choosing motors for robots*, <https://www.engineersgarage.com/egblog/choosing-motor-for-robots/>, Engineers Garage, 2020.
- [37] S. T. Wasim, F. Ahmed, and F. Farooq, “Squadbot: A multi-agent robotics teaching and research platform,” Undergraduate Capstone Report for Habib University, May 2019.
- [38] W. Hess, D. Kohler, H. Rapp, and D. Andor, “Real-time loop closure in 2d lidar slam,” 1998.
- [39] J. M. Santos, D. Portugal, and R. P. Rocha, “An evaluation of 2d slam techniques available in robot operating system,” in *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2013, pp. 1–6.
- [40] M. Sokolov., O. Bulichev., and I. Afanasyev., “Analysis of ros-based visual and lidar odometry for a teleoperated crawler-type robot in indoor environment,” in *Proceedings of the 14th International Conference on Informatics in Control, Automation and Robotics - Volume 2: ICINCO*, INSTICC, SciTePress, 2017, pp. 316–321, ISBN: 978-989-758-264-6.

- [41] R. Simmons, R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan, "A layered architecture for office delivery robots," in *Proceedings of the First International Conference on Autonomous Agents*, ser. AGENTS '97, Marina del Rey, California, USA: Association for Computing Machinery, 1997, pp. 245–252, ISBN: 0897918770.
- [42] J. Kocić, N. Jovičić, and V. Drndarević, "Sensors and sensor fusion in autonomous vehicles," in *2018 26th Telecommunications Forum (TELFOR)*, 2018, pp. 420–425.

## **VITA**

### **Syed Hamza Azeem**

Hamza is an undergraduate Electrical Engineering student in the Class of 2020. He has had an avid interest in solving puzzles and tinkering, since an early age, which is why an engineering education seemed like a natural fit. He has developed an interest in automation and embedded systems through courses like Microcontrollers and Interfacing, Computer Architecture, Mobile Robotics and, Digital Image Processing, as well as a number of projects based around instrumentation, Internet of Things, and hardware-software interfacing. He hopes to use the experience of this Capstone project to explore his interests and find his niche.

### **Sahran Riaz**

Sahran is an undergraduate Electrical Engineering student in the Class of 2020. He possesses a passion for robotics and spends his time following DIY robotics and automaton projects. While having an aptitude for more theoretical endeavors and simulations which is an invaluable asset, he aims to challenge himself in practical matters through the Capstone project, and put into practice his keenness for learning new things. He particularly wants to delve into hardware design, and he also aims to learn new skills, such as, PCB designing, 3D modeling, 3D printing, simulations and experimentation, and embedded systems programming.

### **Sameer Anees Jaliawala**

Sameer is an undergraduate Computer Science student in the Class of 2020. Sameer has always been interested in using software to develop innovative solutions to everyday problems. He has developed his interest through a number of projects involving Database Management, Web Development, and his newfound love for Data Analytics. As such, he wants to use the skills he has gained to work on autonomous systems, and manipulate physical processes through the use software. Sameer has experience with the Robot Operating System framework and Linux which gives him a valuable edge in

realizing the Capstone project. He also has experience in team working in technical environments, as he served 6th in Pakistan in Google's Hash Code competition, and serves as the co-founder of a startup in incubation, YPay.

### **Syed Ahsan Ahmed**

Ahsan is an undergraduate Computer Science student in the Class of 2020. Ahsan discovered his love for Computer Science in his first year at Habib after starting out as an Electrical Engineering major, and he has delved into its various branches with a keen passion. He has developed appreciable experience in Web and Mobile Development through various projects including YPay, a pet project which has now developed into a startup idea being incubated in a leading startup incubator. He hopes to further his experience in Web and Mobile Development, as well as automation through this Capstone project.

### **Sarfraz Shahid Hussain**

Sarfraz is an undergraduate Computer Science student in the Class of 2020. He has been actively involved in a number of Computer Science oriented activities within and outside campus including coding competitions, talks, and workshops. His interest lies towards Data Science and Software Development, and he has consolidated his interest through courses like Data Science, Software Engineering, and Deep Learning. He hopes to build upon his domain knowledge through this Capstone project, and contribute his skills to a project that can potentially be useful for the society.

